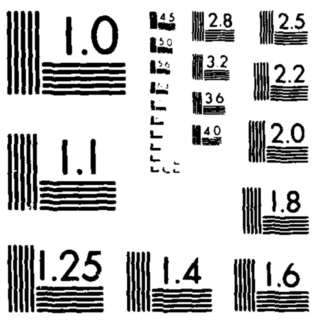
AD-A111 565 FORD AEROSPACE AND COMMUNICATIONS CORP PALO ALTO CA W--ETC F/G 17/2
KSOS SECURE UNIX VERIFICATION PLAN (KERNELIZED SECURE OPERATING--ETC(U)
DEC 80 MDA903-77-C-0333
UNCLASSIFIED WDL-TR7809-REV-2 NL

1 of 1
AD
A111565

END
DATE
FILMED
3-82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU

WDL-TR7809
Revision 2
December 1980

2

ECURE MINICOMPUTER OPERATING SYSTEM (KSOS)

SECURE UNIX VERIFICATION PLAN

Department of Defense Kernelized Secure Operating System

Contract MDA 903-77-C-0333
CDRL 0002AD

Prepared for:

Defense Supply Service - Washington
Room 1D245, The Pentagon
Washington, DC 20310

Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

3939 Fabian Way
Palo Alto, California 94303

NOTICE

The Department of Defense Kernelized Secure Operating System (KSOS) is being produced under contract for the U.S. Government. KSOS is intended to be compatible with the *Western Electric Company's UNIX*™ *Operating System* (a proprietary product). KSOS is not part of the UNIX license software and use of KSOS is independent of any UNIX license agreement. Use of KSOS does not authorize use of UNIX in the absence of an appropriate licensing ageement.

This document, furnished in accordance with Contract MDA 903-77-C-0333, shall not be disclosed outside the Government and shall not be duplicated, used, or disclosed in whole or in part for any purpose other than to evaluate the contractor's performance of Phase I of the contract; upon completion of Phase I of the contract, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the contract.

The contents of this document shall be handled as proprietary information until 5 April 1978. After that time, the Government may distribute the document as it sees fit.

UNIX and PWB/UNIX are trade/service marks of the Bell System.

DEC and PDP are registered trademarks of the Digital Equipment Corporation, Maynard, MA.

Ford Aerospace &
Communications Corporation

KSOS VERIFICATION PLAN

Ford Aerospace/SRI International

3 APRIL 1978

KSOS VERIFICATION PLAN

SECTION I
INTRODUCTION

The purpose of this Verification Plan is to state how the Ford Aerospace and Communications Corp. (FACC) and its subcontractor, SRI International, intend to meet the verification requirements of KSOS, the Kernelized Secure Operating System. Also contained in this document are sections on the mathematical model of security policy to be used in KSOS, on the role of the model, on the programming language to be used for system implementation, on the tools to support the effort, and on the role of testing.

FACC and SRI intend to use a restatement of the Bell and LaPadula model as the conceptual basis for the system security. This modification extends the model of Bell and LaPadula, and incorporates their work as a proper subset. The resulting formulation appears to be well suited to proofs of correspondence between the formal specifications and the formal model. In particular, automatic proof is facilitated, using mostly syntactic properties and being based largely on existing tools. The formulation and the proofs of correspondence are described in more detail in Section II.

One of the properties of the model discussed in Section II is that it has been expressed as constraints on SPECIAL (SPECification and Assertion Language), which is being used as the formal specification language for KSOS. (SPECIAL is described in Roubine and Robinson [77].) Section II also contains a brief example of an illustrative specification in SPECIAL that includes multilevel security and integrity. This example shows how the abstract concepts presented earlier in that section can be applied to a more concrete situation. The example concludes with the theorems derived from the specification, and an indication of how the correspondence proof follows. Most of the theorems are seen to be trivial, lending confidence to FACC's intention to produce complete proofs of the multilevel security of the design.

Section III discusses how the model relates to the KSOS kernel, the emulator, and the user programs. It exhibits the flexibility of the FACC/SRI approach in terms of different security mechanisms and policies that are directly supported by the model, and which can be supported by various implementations consistent with the specifications for the KSOS kernel. Note that only one implementation of KSOS on the 11/70 is planned for Phase II, including the security kernel and the UNIX*tm emulator. However, various alternate emulators can be built on top of the kernel, or in fact the kernel interface can be used directly for other applications. For example, the kernel interface is expected to be suitable for the implementation of a message-processing system instead of the KSOS UNIX*tm emulator. Furthermore, the proofs of correspondence between specifications and the model will apply directly to any other implementation, e.g., by Honeywell (HSOS?) on the SCOMP machine.

The choice of an implementation language for KSOS is discussed in Section IV. Both Euclid and an extended Modula appear to be acceptable candidates, subject to stated assumptions, with Euclid appearing preferable at this time. However, a final decision need not be made until around 1 July 1978, when operational versions of both compilers will be available. This report thus outlines a decision procedure for that language selection, to be made around 1 July 1978. This presents no problems or delays, because the first KSOS coding is not scheduled to take place until late in 1978. Such a strategy presents a minimum risk to the Government, while allowing the advantageous postponement of an important "binding time" until more can be known about the alternatives.

Section V deals with the tools to be used to support the verification of the security of KSOS. These tools can be broken down into two broad categories. The first includes the tools which support the specification process, such as syntactic and semantic analyzers, and related tools to perform the correspondence proofs. The second category includes the tools supporting code to specification checking (program verification). At present the intention is to use existing theorem provers developed under other projects at SRI to produce illustrative but meaningful program proofs. As these other projects are on-going and are supported by diverse funding, additional tools may be available by the time they are needed for KSOS verification. We intend to use the best available technology in providing the Government with a high confidence system. FACC and SRI also plan to monitor developments at other research centers for their potential applicability to the KSOS verification effort.

Section VI discusses the testing to be performed on KSOS. It is appropriate to include this material here because testing and verification are both ways of increasing confidence in the satisfaction of critical system properties. FACC feels that testing and verification are mutually supportive. Measures taken to improve the verifiability of the product enhance its ease of testing. (Testing is further discussed in the companion KSOS Implementation Plan.)

REFERENCE

Roubine and Robinson [77] O. Roubine and L. Robinson, SPECIAL (SPECification and Assertion Language): Reference Manual, SRI Technical Report CSG-45, 3rd Edition (January 1977).

SECTION II
MULTILEVEL SECURITY MODEL FOR KSOS

INTRODUCTION

To prove security properties of a system design, it is necessary to formulate a mathematical model of the desired security. It is important that this model be easily related to some formal description of the system design in order that the proof be feasible. However, it is equally important that the mathematical model be simple and easily comprehensible. Users of the system must be convinced that the model represents their intuitive conception of security. If a belief in the security of systems is to be widely held, it is necessary that the successive stages of:

1) intuitive notion of security,
2) mathematical model of security,
3) secure system design, and
4) secure system implementation

be related to one another in a straightforward manner. This paper presents mathematical models of multilevel security that significantly improve upon existing models of multilevel security in achieving this goal.

One model that approximates the military security needs has been developed by Bell and LaPadula [74], and has been applied to the design and verification of a security kernel (Millen [76]). A similar model has been described by Walter et al. [75]. These models describe systems in which information may pass from repositories of one security level to repositories of only the same or higher security level. This report presents a multilevel security model that is a generalization of the Bell and LaPadula and Walter models, and reformulates this generalized model in terms more amenable to proof. The concern here is for proving properties of system specifications. The implementation of the system must also be proved correct with respect to the specification, but this issue is not within the scope of this report.

In the Bell and LaPadula model each SUBJECT (i.e., a process operating on behalf of a user) is assigned a unique SECURITY LEVEL (or simply LEVEL). A subject can create (activate) an OBJECT, which then acquires its own security level, namely that of the subject. The subject is then free to operate on the object as he wishes, including deleting (deactivating) it. It is useful to view the operations as in two classes: MODIFY and READ. The Bell and LaPadula model consists of the following five axioms:

SIMPLE SECURITY CONDITION -- A subject S can read an object OB only if the security level of S is at least that of OB. This condition allows a subject to read only from objects whose security level is less than or equal to his level.

*-PROPERTY (pronounced "star property") -- A subject can modify an object OB1 in a manner dependent on data in an object OB2 only if the security level of OB1 is at least that of OB2. This condition prevents a subject from transferring information from OB2 to OB1 unless OB1's level is at least that of OB2.

TRANQUILITY PRINCIPLE -- A subject cannot change the security level of an active object.

NON-ACCESSIBILITY of INACTIVE OBJECTS -- A subject cannot read the contents of an inactive object.

REWRITING OF NEWLY ACTIVATED OBJECTS -- A newly activated object is given an initial state that is independent of the state of any previous incarnations of the object.

The fact that these properties imply a multilevel secure system is not immediately obvious. Nor are these properties ideally suited to proving that a given design is multilevel secure. This paper presents two models of multilevel security that together fulfill the goals of comprehensibility and provability. The second of the two models presented is essentially equivalent to the Bell and LaPadula model, but is formulated in a manner that makes the proof straightforward that a given system design is multilevel secure. The first model presented is more general and more abstract, thereby making it easily comprehensible to a casual reader. The proof that the second model is actually a restriction of the first is straightforward.

MULTILEVEL SECURITY

Each user of the system has one or more independent processes operating solely on his behalf. Each process has associated with it a CLEARANCE and a CATEGORY SET. The system has a fixed finite number of clearances that are totally ordered by the relation "less than". For example, the clearance CONFIDENTIAL is less than SECRET, which is less than TOP SECRET. For convenience, clearances are represented as integers.

A category set is any subset of the set of all possible categories. Examples of categories might be ATOMIC and NATO. The combination of a clearance and a category set is called a SECURITY LEVEL or equivalently ACCESS LEVEL; for simplicity, it is often called just a LEVEL when ambiguity is not likely to arise. A security level L1 is equal to a security level L2 if and only if the clearance of L1 is equal to the clearance of L2 and the category set of L1 is equal to the category set of L2. A security level L1 is said to be less than or equal to a security level L2 whenever the clearance of L1 is less than or equal to the clearance of L2, and the category set of L1 is a subset of the category set of L2. L1 is less than L2 whenever L1 is less than or equal to L2 and L1 is not equal to L2. Thus the set of all security levels can be partially ordered. Note that not all security levels are related by the partial ordering, e.g., two processes with respective security levels <SECRET, {ATOMIC}> and <SECRET, {NATO}> are not comparable. The security levels and the relation "less than" define a lattice since there is a minimum and maximum clearance, and a maximum set of categories.

In informal terms, a system is MULTILEVEL SECURE if and only if, for any two processes P1 and P2, unless the security level of P1 is less than or equal to the security level of P2, there is nothing that P1 can do to affect, in any way, the operation of P2. That is, P2 is not able to know anything about P1, not even the existence of P1. This constraint implies that P1 cannot affect the operation of P2 using an intermediate process P3. It is not possible for a process at a higher level to transmit information to a process at a lower level. Therefore, INFORMATION CAN ONLY FLOW UPWARD IN SECURITY OR REMAIN AT THE SAME LEVEL, i.e, can only flow to processes of greater or equal security level.

The above constraint is consistent with the real military security situation, since -- for example -- an individual whose category set contains only ATOMIC cannot pass information to an individual whose category set does not contain ATOMIC, independent of the latter's clearance or the other components of his category set.

For each user in the system, there is a maximum security level at which he can operate, e.g., as the result of the login routine assigning a process at that level to execute on his behalf. The user can select to operate at one or more security levels less than or equal to this maximum level. A process at each of the chosen security levels will be created to operate on the user's behalf at that particular level. However, the existence of these processes may be hidden from the user by a suitable interface program such as SIGMA (Ames and Oestreicher [77]). (It should be noted that the model and the system design are not intended to prevent a user from generating TOP SECRET information when logged in at CONFIDENTIAL, or from showing a TOP SECRET document to an uncleared colleague when off-line. This behavior is outside the scope of the model and is not under the control of the computer system.)

The multilevel security model does not prohibit a process at some security level from modifying information at a higher security level. However, there are many cases in which such a prohibition is desirable. A group at MITRE (Biba [77]) has identified the concept of integrity to solve this problem. Integrity is the precise formal dual of multilevel security. In addition to a security level, each process of the system has an associated integrity level. The set of integrity levels is identical to the set of security levels and has the same relation "less than". A system has multilevel integrity if and only if, for any two processes P1 and P2, unless the integrity level of P1 is greater than or equal to the integrity level of P2, there is nothing that P1 can do to affect, in any way, the operation of P2. Therefore, information can only flow downward in integrity or remain at the same integrity level. Integrity can be used to limit the upward flow of information enforced by multilevel security. It is important to remember that a process's security level and its integrity level need not be the same. The primary advantage of using integrity as a further means of restricting information flow is that, being the formal dual or security, it adds no significant complexity to the security model and no significant complexity to the proof of a secure system design.

GENERAL MODEL OF MULTILEVEL SECURITY AND INTEGRITY

A system consists of a collection of operations or functions. Each function may be invoked by a user of the system (actually the function is invoked as part of a program running on behalf of a user). When invoked, a function may take a set of arguments. A function together with a particular set of arguments is termed a function reference. When a function reference is invoked, it can cause the state of the system to change and/or return information to its invoker. The set of all function references of a system is called F and some member of this set is denoted by f.

We also define a set of security and integrity levels L. The security and integrity levels L are partially ordered by the relation "<". Multilevel security involving classifications and categories, is but one example of a partial ordering of security and integrity levels, so we will be dealing here with a more general case. There are functions K and I whose domain is F and whose range is L. The functions K and I return respectively the security and integrity levels of their argument. A process is assigned a security level and an integrity level for its lifetime and may only invoke function references at these levels. (Note that a user may have several processes operating on his behalf simultaneously, and may therefore operate at several security and integrity levels.)

Finally, we introduce the relation "-->" on function references. We say that

$$f_1 \dashrightarrow f_2$$

(read as f_1 transmits information to f_2) if there is any possibility that the information returned by an invocation of f_2 could have been in any way effected by a prior invocation of f_1. In other words, there is some transmission of information from f_1 to f_2.

The definition of multilevel security can now be stated simply. For any f_1 and f_2 in F:

$$\boxed{f_1 \dashrightarrow f_2 \implies K(f_1) <= K(f_2) \text{ AND } I(f_1) >= I(f_2) \qquad (P1)}$$

This simply states that if there is any possibility of information transmission between two function references, then the transmitting function reference must have a security level less than or equal to the that of receiving function reference, and the receiving function reference must have an integrity level less than or equal to that of the transmitting function reference.

In other words, information can only flow upward in security or remain at the same level. A more formal definition is given in the appendix. Similarly, information can only flow downward in integrity or remain at the same level.

Unfortunately, the abstract nature of this definition makes it difficult to relate to constructs used in expressing system designs. This gap can be bridged by formulating a slightly more restrictive model in less abstract terms.

RESTRICTED MULTILEVEL SECURITY MODEL

Each state variable v contains some of the state information of the system. The state variables together completely describe the state of the system. The value of each state variable may be modified by invocation of some function reference. Each state variable is assigned a security level and an integrity level which is determined by extending the functions K and I to apply to state variables as well as function references, therefore, K(v) is the security level of state variable v and I(v) is the integrity level of state variable v. The relation $\xrightarrow{f}$ relates two state variables such that

$$v_1 \xrightarrow{f} v_2$$

means that an invocation of function reference f may cause the value of v_2 to change in a manner dependent upon the previous value of v_1. In other words there is an information flow from v_1 to v_2 caused by the invocation of f. Two predicates must also be defined: the prefix form of $\xrightarrow{f}$

$$\xrightarrow{f} v$$

means that an invocation of the function reference f may cause the value of state variable v to change; the postfix form

$$v \xrightarrow{f}$$

means that the value returned by function reference f is dependent on the prior value of state variable v. Note that for any f_1, v_1, and v_2:

$$v_1 \xrightarrow{f} v_2 \implies \xrightarrow{f} v_2$$

A multilevel secure system may now be redefined. For any function reference f and state variables v, v_1, and v_2

P2

$$v \xrightarrow{f} \implies K(v) \le K(f) \text{ AND } I(v) \ge I(f) \qquad \text{(P2a)}$$

$$v_1 \xrightarrow{f} v_2 \implies K(v_1) \le K(v_2) \text{ AND } I(v_1) \ge I(v_2) \qquad \text{(P2b)}$$

$$\xrightarrow{f} v \implies K(f) \le K(v) \text{ AND } I(f) \ge I(v) \qquad \text{(P2c)}$$

These properties assure that information flow is always upward in security level, downward in integrity level, or remains at the same security or integrity level. Loosely speaking, the arrow --> always points upward in security level and downward in integrity level. P2a states that the value returned by an invocation of a function reference at some security and integrity levels contains information from state variables at only lower or equal security levels or higher or equal integrity levels. P2b assures that when information is transferred from one state variable to another by some invocation of a function reference, that the recipient variable is at a higher or equal security level or lower or equal integrity level than the originator variable. P2c assures that the value of a state variable may be changed by invocation of a function reference whose security level is less than or equal to or whose integrity level is greater than or equal to that of the variable, thereby guaranteeing that security cannot be violated by the act of invoking a function reference. A more formal model of properties P2 is given in the appendix.

If state variables are equated to objects in the Bell and LaPadula model, then properties P2 are approximately equivalent to the simple security and integrity properties and the *-property for security and integrity. In place of defining reading and writing of state variables, we use functional dependency. The new value of state variable (object) A being functionally dependent upon the prior value of state variable B is similar to saying that A is written and B is read. In many cases the simple notion of a state variable being read or written is not sufficient because it is important to know how the value of that state variable effects the values of other state variables (i. e., how the information is flowing).

There is one fundamental difference between the Bell and LaPadula model and the properties P2. Bell and LaPadula consider only operations that either read a value from an object or modify the value of an object. Operations that both read a value from an object and modify the value of an object can be obtained by the composition of operations. The model presented above treats operations that both read values and modify values as primitive. This added consideration makes the properties P2 slightly more complex but also makes the properties slightly more general than the simple security property and *-property of Bell and LaPadula. If the case of operations that could both read and modify were removed from the model, then property P2a would correspond to the simple security property (and simple integrity property), property P2c would correspond to the *-property, and property P2b would be unnecessary.

RELAXATION OF THE MODEL

Although the general definition of multilevel security presented above is quite simple, it does not precisely model the real world of military multilevel security. The model does not take into account, for example, declassification of objects or objects consisting of other objects with different classifications. This same deficiency is true of the Bell and LaPadula model. In order to deal with these necessary functions, it is useful to consider some small changes to the model that permit the additional capabilities. It is essential that these changes be both small, in order that the model retain its simplicity, and meaningful, so that the model is easy to interpret. The proposed new functions require relaxations of the model to some extent because the new functions violate the existing model. For the remainder of this section, the model represented by properties P2 will be termed the strict model to distinguish it from the relaxed versions to be presented below.

Rather than formulating a distinct modification to the model for each new function, it is more desirable to try to find a few modifications that cover many cases. The most obvious such modification corresponds to removing the *-property from the Bell and LaPadula model, leaving only the simple security property. Although, for reasons stated above, there is no direct equivalent to the simple security property in the strict model, there is a property that is equivalent. Before presenting this property, it is necessary to reformulate properties P2 as follows:

<u>P2'</u>

$$v \xrightarrow{f} \implies K(v) \le K(f) \text{ AND } I(v) \ge I(f) \qquad \text{(P2a)}$$

$$v_1 \xrightarrow{f} v_2 \implies \qquad \text{(P2b')}$$
$$(K(f) \le K(v_1) \implies K(v_1) \le K(v_2))$$
$$\text{AND } (I(f) \ge I(v_1) \implies I(v_1) \ge I(v_2))$$

$$\xrightarrow{f} v \implies K(f) \le K(v) \text{ AND } I(f) \ge I(v) \qquad \text{(P2c)}$$

The only property changed is P2b. These properties are equivalent to properties P2. This modified set of properties will be called P2'. The equivalent of the simple security and simple integrity properties is the combination of P2a and P2b'. P2a states that reading up is not permitted. P2b' states that downgrading is permitted for state variables whose security level is less than or equal to the security level of the operation or for state variables whose integrity level is greater than or equal to the integrity level of the operation. The *-properties for security and integrity are equivalent to the combination of P2b' and P2c. This means that the enforcement of either the simple security properties or *-properties can be removed by removing properties P2a or P2c respectively from this model. Note that by discarding the *-property, the extent of the security is changed: there is no longer confinement, and Trojan horses are possible; however, there is still a meaningful form of security and the simplicity of the model has not been compromised.

Simply removing the *-property from the security model relaxes the strict multilevel security constraints to permit many of the desired additional capabilities. However, one can anticipate needs for which the simple security and integrity properties are still too restrictive. For these cases one can permit a process to violate either multilevel security or integrity or both. However, for these cases it is assumed that such a process will enforce some other type of security that can be verified. Therefore, full relaxation of multilevel security should never be perceived by a user of the system. Such extreme relaxation of the multilevel security restraints should be viewed as an internal mechanism which allows the system to support a variety of security policies.

Each process in the system must obey one of the constraints from each of the following columns (i.e., one from column A and one from column B):

Column A	Column B
Strict multilevel security	Strict multilevel integrity
Simple security property	Simple integrity property
No multilevel security	No multilevel integrity

Proving that the specification for any given operation is consistent with any of the above requirements is no more difficult than proving that the specification is consistent with strict multilevel security and integrity. Such proofs are the topic of the next section.

SPECIFICATIONS AND PROOF TECHNIQUE

For the purpose of writing specifications we use the language SPECIAL (SPECIfication and Assertion Language, Roubine and Robinson [77]). In SPECIAL, the visible functions of a system design are partitioned into two types:

V-functions - return information about the state of the system but does not change the state of the system,

OV-functions (including what is called O-functions) - change the state of the system and may return information about the state.

The actual state of the system is described by the "primitive" V-functions, i. e., functions that return the value of a particular state variable of the system. The primitive V-functions are descriptive artifacts of the specifications and need not be present in an implementation. The value of a primitive V-function may be available to a user of the system if there is a visible V-function that returns the value of the primitive V-function. The values returned by visible V-functions are functions of the values of only the primitive V-functions.

The specification of each visible function has two major parts. The first part is the EXCEPTIONS, a list of boolean valued expressions. If any of these expressions evaluates to true for a given invocation of a function, then the function is aborted with no change of state to the system. The values of these exceptions are results of the function invocation since the occurrence of an exception is reported to the caller of the aborted function.

For a visible V-function, the second part of the function specification is the DERIVATION, an expression whose value is the result of the V-function. The value is returned only if all the exceptions of the V-function invocation are false. For an OV-function, the exceptions are followed by the EFFECTS, assertions that relate the values of the state variables (primitive V-function references) subsequent to the invocation of the OV-function to the values of the state variables prior to the invocation of that OV-function. Subsequent values of state variables are denoted in effects by preceding the primitive V-function references corresponding to those state variables by a single quote ('). Prior values are unquoted.

Note that there is a very strong correlation between the model underlying the semantics of SPECIAL and the model of a system used to describe the strong multilevel security properties, P2. The state variables of the security model are references of the primitive V-functions of SPECIAL and the function references, F, of the security model are references of the visible functions of SPECIAL. The values of function references of the security model are the return values and exceptions of the visible functions in SPECIAL. We have also added a convention that prescribes that each primitive function reference of a SPECIAL specification contain a formal parameter that is the security and integrity levels of that function reference. For visible V-functions, the security and integrity levels of a function reference are enclosed in square brackets ([...]) after the formal parameter list. The properties P2a, P2b, and P2c can, therefore, be directly applied to specifications written in SPECIAL. Illustrations of how properties P2a, P2b, and P2c can be applied to specifications for purposes of proof are given in Feiertag et al. [77] and Neumann et al. [77].

There are two difficulties that make proof of the consistency of the specifications and the properties P2 nontrivial. First, the specifications are written in terms of function descriptions, not function reference descriptions. This means that one must prove that the properties P2 hold for all possible arguments to the functions described in the specifications. In many cases some sets of arguments to a particular function must be considered as distinct cases in order to make the proof tractable. The appropriate partitioning of cases requires careful judgment. Second, in describing the change of state caused by an OV-function invocation, SPECIAL permits considerable freedom in expressing the relation between the new values of the primitive V-function references and their prior values. The use of recursive functions and universal and existential quantifiers makes it undecidable in general to determine if a new value of a primitive V-function reference is functionally dependent upon the prior value of some other primitive V-function reference. Since functional dependency is generally undecidable, we have derived a set of decidable dependency rules that are used to determine if the value of some quoted primitive V-function reference (new value of a state variable) is functionally dependent upon some unquoted primitive V-function reference (prior value of a state variable) for the most common of the decidable cases. When these rules cannot be definitively applied, a functional dependency is assumed. These rules are similar to the elimination rules of Millen [76]. For the specifications we have examined, we have had no difficulty in deriving an acceptable set of such rules. The example given later illustrates the proof technique and utilizes a particularly simple set of these decidable dependency rules.

EXAMPLE

In order to illustrate the proof technique, a proof of a representative operation will be presented. Fig. 2.1 is a specification of the module "files" that is part of the KSOS emulator. As this module was written during the early design stages, it is subject to modification and and may not appear in similar form in the final design. However, the module may be considered representative in style, size, and complexity of modules in the KSOS design, being perhaps a little simpler than most. The proof of security of the O-function "write" will be examined. The proof of properties P2a, P2b, and P2c require the identification of all instances of primitive V-function references within the operation to be proved. Many such instances are enclosed in the macro facilities of SPECIAL (namely the DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF) so those macro definitions containing primitive V-function references are expanded yielding the specification for the "write" operation given in Fig. 2.2. (Note that in some cases of definitions, it is possible to assign a security level to the definition itself and treat the definition as a primitive V-function. This shortens the proof process somewhat. However, this technique will not be discussed here).

Each function reference must be assigned a security and integrity level, collectively called an access level. In order to guarantee that the levels of function references do not change (a requirement of the multilevel model), one of the arguments to each function reference will be its access level. By convention, the access level argument will be the formal parameter in the definition of the function that is named "level". The relation "<=" is defined for access levels by the definition "write_allowed" and the relation ">=" is defined for access level by the definition "read_allowed". In order to avoid repeating these definitions in all modules of the specifications, the definition of access level and its associated relations will be encapsulated into a separate module in the final specifications.

The next step in the proof process is to generate a set of theorems whose validity implies properties P2a, P2b, and P2c. The theorems generated for the "write" operation are given in

Fig. 2.3. These theorems are derived from the specifications using knowledge of the syntax of SPECIAL and the decidable dependency rules (which embody the semantics of SPECIAL). An examination of two of these theorems serves to illustrate the theorem-generating step of the proof process. Properties P2a, P2b, and P2c must be proved for each visible function reference, i.e., the proof must be carried out for all possible set of arguments, hence the universal quantification of all the arguments in each theorem. Consider the first theorem of Fig. 2.3. This theorem is part of the proof of property P2a which states that the value returned by a visible function reference can be dependent upon only the values of primitive V-function references at lower security levels. Since the occurrence of an exception is a returned value, all the primitive V-function references cited in exceptions must be considered. The first theorem of Fig. 2.3 deals with the first exception which cites the primitive V-function i_nlink. The level of all references to the visible function "write" is "level", and the level of all references to this citation of i_nlink is "l". The relationship to be established is that "level" is greater than or equal to "l" or, stated in terms of the definitions, that read_allowed(level, l) is true. This condition is the stated consequent of this theorem. The value of the SOME construct in this expression is potentially dependent upon all possible "l" because the SOME construct implies a universal quantification, hence the universal quantification of "l" in the theorem (a decidable dependency rule). However, the value of the exception is dependent on values of i_nlink only when read_allowed(level, l) is true. This follows because read_allowed(level, l) is a conjunct of i_nlink(fid, l) and when the former is false, the value of the latter is irrelevant. This fact results in the qualification in the theorem (another decidable dependency rule). This latter rule can be stated more explicitly as: when an expression citing no primitive V-function references is a conjunct to an expression citing some primitive V-function references, then the former expression will always be, in the generated theorems, an antecedent to any consequents evolving from the latter expression

Consider now the third theorem of Fig. 2.3. Again note the universal quantification over the arguments to the visible function "write" due to the necesity of proving the theorem for each visible function reference. This theorem is used as part of the proof of property P2b, which states that the level of each quoted primitive V-function reference must be greater than or equal to the level of each unquoted primitive V-function reference upon which it is dependent. This particular theorem deals with the dependency of the value of the quoted reference to h_file upon the value of the unquoted reference to h_file in the effects. Note that there are two unquoted references to h_file in the effects and that in general each would necessitate a separate theorem, however, in this case the two necessary theorems happen to be identical. Note that the antecedent in this theorem is actually the conjunction of the negation of the two exceptions. This is because the semantics of SPECIAL demand that the exceptions be false for the effects to occur (another decidable dependency rule). The consequent of this theorem simply expresses the desired relationship between the levels of the quoted and unquoted primitive V-function references to h_file.

The first two theorems of Fig. 2.3 are necessary to prove property P2a, the next two theorems are for P2b, and the final theorem is for P2c. The proofs of the first three theorems are trivial, as they can be shown true directly from the axioms of the logical operations with no deduction. The last two theorems require only minimal amounts of deduction. These theorems are representative of most theorems generated using this technique. They indicate that only very simple theorem proving capability is necessary and that the automation of the theorem-proving step of the proof process (the final step) is desirable.

A simple upper bound can be placed on the number of theorems generated for a given visible function. Using the following definitions:

nxv = the number of citations of primitive V-functions in the exceptions

nqv = the number of citations of quoted primitive V-functions in the effects

nuv = the number of citations of unquoted primitive V-functions in the effects

the number of theorems generated will be at most

$$nxv + (nqv + 1) * nuv + nqv$$

For the "write" operation this upper bound is 13, whereas the actual number of theorems is 5. In this case the failure to reach the upper bound is due to the absence of a return value (other than the exceptions) and that some of the theorems happen to be identical and have not been replicated.

It is important to realize that this particular example is probably simpler than the proof of a typical visible function in a system such as KSOS. A more representative example is likely to contain more DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF expressions that contain citations of primitive V-functions thereby yielding a much greater number of such citations in the expanded form of the function specification, hence a much greater number of theorems. In fact, the listing of theorems is undoubtedly going to be much longer than the listing of the specifications from which the theorems are derived. The saving grace is that the proof of the theorems are rather simple and are amenable to automation. In addition, the use of the decidable dependency rules makes generation of the theorems from specifications amenable to automation as well. The entire proof of design could therefore be automated.

However, this automation depends, in part, on the discovery of a suitable set of decidable dependency rules. A suitable set of rules sufficiently complete to allow generation of theorems that can be easily proved depends upon the nature of the system to be proved and the specification writing style of the authors of the specifications. Experience with the proof technique will enable us to arrive at such a sufficiently complete set of rules that are applicable to KSOS as well as other systems.

ADDITIONAL CONSIDERATIONS

The problem of using time as an illicit information channel has plagued the designers of secure operating systems for many years. The specification of the FILES module given above does not mention time at all, even though it is always an observable piece of information in real systems. This means that the specifications do not fully describe the implementation. The concept of time could be introduced into the specifications. Unfortunately, in order to do this and still be consistent with the multilevel security model, the resulting design yields an inherently inefficient system. This same fundamental problem has been identified by Lampson [73], Lipner [75], and Millen [76].

We have illustrated in an informal manner how to prove the security of a system design. The formal proof is quite straightforward but requires many steps, and is quite lengthy. In general, the formal proof of multilevel security of a given set of specifications will be longer than the specifications themselves. This raises the possibility that the proof may be more error-prone than the specifications. It is, therefore, desirable to automate the proving process. The verification conditions that must be generated by an automated prover to prove properties P2a, P2b, and P2c can be derived from the specifications, the syntax of SPECIAL, and the rules for potential dependency. The verifications thus generated, although lengthy, are quite simple, and require only a fairly unsophisticated theorem prover. In some cases, human intervention may be required to supply lemmas to aid the theorem prover. A proof checker program can be included to double check the validity of the output of the theorem prover. If proof of security is to become a meaningful way to verify systems, some form of automation is essential.

CONCLUSIONS

We have presented two formal definitions of multilevel security. The first model is a generalization and abstraction of the Bell and LaPadula model, the Walter model, and the second model presented here. Each of the two models is particularly well suited to specific needs of the design and proof process. The first model, besides providing a more general definition of multilevel security, is very simple, enabling the reader to verify easily that the definition is consistent with intuitive understanding of multilevel security. The second model relates directly to techniques used to specify systems. It is easy to prove the correspondence between the specifications of a system and this model. This proof technique can be automated, a step that is essential in enhancing the credibility of the proof. We have demonstrated the generality and simplicity of the general model and have described the technique used to prove the consistency of specifications in SPECIAL with the restricted model. It should be possible to automate this proof technique. This multiple model approach, i. e., models describing the concept at various levels of abstraction, appears to be advantageous in providing both a simple definition (the first model) that can be easily related to intuitive notions of multilevel security, and a set of principles (the second model) that can be directly related to the design of a multilevel secure system.

The security proof of a system design is only one part of proving the security of an actual running system. However, proving the security of the design alone, before implementation is attempted, can be far more cost effective than discovering the security flaws during the implementation or operation of the system, or having an insecure system.

APPENDIX
FORMAL MODELS

A multilevel system is defined to be the following ordered 10-tuple:

$$\langle S, s_0, L, "<", F, K, I, R, N_r, N_s \rangle$$

where the elements of the system can be intuitively interpreted as follows:

S - States: the set of states of the system

s_0 - Initial state: the initial state of the system; $s_0 \in S$

L	- Security levels: the set of security levels of the system
"<"	- Security relation: a relation on the elements of L that partially orders the elements of L
F	- Visible function references: the set of all the externally visible functions and operations (i.e., functions and operations that can be invoked by programs outside the system); if a function or operation requires arguments, then each function together with each possible set of arguments is a separate element of F (note that in the remainder of this document externally visible functions and operations will be referred to collectively as visible functions (or functions) even though operations are not functions in the mathematical sense)
K	- Function reference security level: a function from F to L giving the security level associated with each visible function reference; a process may invoke only function references at the security level of the process; K:F->L
I	- Function reference integrity level: a function from F to L giving the integrity level associated with each visible function reference; a process may invoke only function references at the integrity level of the process; I:F->L
R	- Results: the set of possible values of the visible function references
N_r,N_s	- Interpreter: functions from F$\times$S to R and S that define how a given visible function reference invoked when the system is in given state produces a result and a new state; N_r:F$\times$S->R and N_s:F$\times$S->S.

In order to define multilevel security and integrity, the following definitions are useful:

T - the set of all n-tuples of visible function references or, in other words, all possible sequences of operations

$$T = F^*$$

M - the function whose value is the state resulting from the given sequence of operations starting at some given state

M:S$\times$T->S

D - the function whose value is the set of state variables whose values differ in the given states

$D: S \times S \to V^*$

E_1 - the function whose value is the sequence of operations that results when all the operations whose level is not less than or equal to the level "1" are removed from the given sequence of operations

$E_1 : T \to T$

E^1 - the function whose value is the sequence of operations that results when all the operations whose level is not greater than or equal to the level "1" are removed from the given sequence of operations

$E^1 : T \to T$

Multilevel security and integrity can now be defined as follows:

$$\boxed{\begin{array}{l} (\forall f \in F, s \in S, t_1, t_2 \in T) \\ E_{K(f)}(t_1) = E_{K(f)}(t_2) \text{ AND } E^{I(f)}(t_1) = E^{I(f)}(t_2) \qquad (P1) \\ \Longrightarrow N_r(f, M(s,t_1)) = N_r(f, M(s,t_2)) \end{array}}$$

Informally, if two sequences of operations are each applied to a system in the same state and if these sequences differ only in operations whose security level is not less than or equal to some level, then any operation of that level that is invoked immediately following either of the two sequences will return the same result. In other words, the operations whose security level is not less than or equal to this level cannot effect results visible to the level. For integrity, the interpretation is that operations whose integrity level is not greater than or equal to this level cannot effect results visible to the level.

This formal definition simply expresses the essence of the informal definition given above: information can only flow upward in security or remain at the same level. The formal property, P1, expresses this definition in terms of only the externally visible behavior of the system without regard to invisible internal properties. There is no need to identify objects, distinguish read and write operations on objects, or assign security levels to objects in this definition as is done in the Bell and LaPadula and the Walter models. It is the simplicity of this formal definition that makes it easily comprehensible to the reader.

The state of a system is determined by the values of the state variables of the system. That is, the state of the system is the cross product of all the state variables of the system where each state variable is a set of values that can be assumed by the state variable. The function G(V) defines the security level to which the state variable V is assigned, and H(V) defines the integrity level of V. The following useful functions can now be defined:

$P_1 : S \rightarrow \underset{\forall V \mid G(V)=1}{\mathsf{X}} V$	is a function that takes a state to that part of the state consisting of the ordered tuple of values of state variables assigned to security level 1
$P^1 : S \rightarrow \underset{\forall V \mid H(V)=1}{\mathsf{X}} V$	is a function that takes a state to that part of the state consisting of the ordered tuple of values of state variables assigned to integrity level 1
$Q_1 : S \rightarrow \underset{\forall V \mid G(V)<=1}{\mathsf{X}} V$	is a function that takes a state to the tuple of values of state variables assigned to a security level less than or equal to 1
$Q^1 : S \rightarrow \underset{\forall V \mid H(V)>=1}{\mathsf{X}} V$	is a function that takes a state to the tuple of values of state variables assigned to an integrity level greater than or equal to 1
$W_1 : S \rightarrow \underset{\forall V \mid \neg(G(V)>=1)}{\mathsf{X}} V$	is a function from a state to the tuple of values of state variables assigned to a security level not greater than or equal to 1

$W^{l}: S \rightarrow \underset{\forall V | \neg (H(V) <= l)}{\times} V$ is a function from a state to the tuple of values of state variables assigned to an integrity level not less than or equal to l

It is now possible to define three new security properties whose conjunction is somewhat stronger than P1 above:

<u>P2</u>

$(\forall f \in F)\ (\exists a)(\forall s \in S)$ (P2a)

$$N_r(f,s) = a(Q_{K(f)}(s),\ Q^{I(f)}(s))$$

$(\forall f \in F, l \in L)\ (\exists a,b)(\forall s \in S)$ (P2b)

$$P_l(N_s(f,s)) = a(Q_l(s)) \text{ AND } P^{l}(N_s(f,s)) = b(Q^{l}(s))$$

$(\forall f \in F, s \in S)$ (P2c)

$$W_{K(f)}(s) = W_{K(f)}(N_s(f,s)) \text{ AND } W^{I(f)}(s) = W^{I(f)}(N_s(f,s))$$

where "a" and "b" represent arbitrary functions. The first property (P2a) states that the result of a function invocation can be functionally dependent only upon state variables of security level lower than or equal to the security level of the function reference and integrity level greater than or equal to the integrity level of the function reference. The second property (P2b) states that the value assumed by a state variable at some security and integrity level due to the action of some function invocation can be functionally dependent only upon state variables at a lower or equal security level and greater or equal integrity level respectively. The third property (P2c) states that the values of state variables at some security and integrity level can be changed only by function invocations at a lower or equal security level and greater or equal ir :egrity level respectively. The proof that properties P2a, P2b, and P2c imply property P1 is done by induction on the length of the sequence E(t, l) for each level l and sequence of visible function references t. The proof is described in Neumann et al. [77].

As an alternative to these definitions, the relations and predicates used to describe the model informally in the body of the plan above, can be described precisely to yield a formal definition of security. The two relations and two predicates described above can be formally defined as:

$$\boxed{\begin{array}{l} f_1 \dashrightarrow f_2 \quad \langle == \rangle \\ (\exists t_1, t_2 \in T) \\ \quad N_r(f_2, M(t_2, M(\langle f_1 \rangle, M(t_1, s_0)))) \\ \quad {\sim=}\ N_r(f_2, M(t_2, M(t_1, s_0))) \end{array}}$$

$$\boxed{\begin{array}{l} v_1 \xrightarrow{f} v_2 \quad \langle == \rangle \\ (\exists s_1, s_2 \in S \mid D(s_1, s_2) = \{v_1\}) \\ \quad v_2 \in D(N_s(f, s_1), N_s(f, s_2)) \end{array}}$$

$$\boxed{\begin{array}{l} v \xrightarrow{f} \quad \langle == \rangle \\ (\exists s_1, s_2 \in S \mid D(s_1, s_2) = \{v\}) \\ \quad N_r(f, s_1) \ {\sim=}\ N_r(f, s_2) \end{array}}$$

$$\boxed{\begin{array}{l} \xrightarrow{f} v \quad \langle == \rangle \\ (\exists s \in S) \\ \quad v \in D(s, N_s(f, s)) \end{array}}$$

REFERENCES

Ames and Oestreicher [77] S. R. Ames and D. R. Oestreicher, Design of a Message Processing System for a Multilevel Secure Environment, unpublished (October 1977).

Bell and LaPadula [74] D. E. Bell and L. J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, MITRE Corp., Bedford MA (September 1974).

Biba [77] K. J. Biba, Integrity Considerations for Secure Computer Systems, MTR-3153 Rev. 1, MITRE Corp., Bedford MA (April 1977).

Feiertag et al. [77] R. J. Feiertag, K. N. Levitt, and L. Robinson, Proving Multilevel Security of a System Design, Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Purdue Univ., West Lafayette, IN (November 1977).

Lampson [73] B. W. Lampson, A note on the confinement problem, CACM vol. 16 no. 10, pp. 613-615 (October 1973).

Lipner [75] S. B. Lipner, A comment on the confinement problem, The MITRE Corporation, Bedford MA (1975).

Millen [76] J. K. Millen, Security Kernel Validation in Practice, CACM vol. 19 no. 5, pp. 243-250 (May 1976).

Neumann et al. [77] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt,L. Robinson, A Provably Secure Operating System: the System, its Applications, and Proofs, Stanford Research Institute, Menlo Park CA (February 1977).

Roubine and Robinson [77] O. Roubine, L. Robinson, SPECIAL Reference Manual, 3rd edition, Stanford Research Institute, Menlo Park CA (January 1977).

Walter et al. [75] K. G. Walter, W. F. Ogden, J. M. Gilligan, D. D. Schaeffer, S. I. Schaen, D. G. Shumway, Initial Structured Specifications for an Uncompromisable Computer Security System, Case Western Reserve University, Cleveland OH (July 1975).

```
MODULE files

    TYPES

clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);


    DECLARATIONS

file_id fid;


    PARAMETERS

INTEGER super_user_id;
INTEGER max_clearance, number_of_categories;


    DEFINITIONS

BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
       >= object_al.security_clearance
      AND subject_al.integrity_clearance
         <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
            ( object_al.security_categories[i]
             => subject_al.security_categories[i])
           AND( subject_al.integrity_categories[i]
               => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
access_level read_level(fid; access_level level)
   IS SOME access_level l |
       read_allowed(level, l) AND i_nlink(fid, l) > 0;
BOOLEAN is_file_owner(fid; INTEGER uid; access_level level)
   IS (uid = i_uid(fid, read_level(fid, level)))
      OR (uid = super_user_id);
BOOLEAN file_exists(fid; access_level level)
   IS read_level(fid, level) ~= ?;
```

Fig. 2.1 - Specification of files (continued on next page)

```
    EXTERNALREFS


    FROM metafiles:
VFUN i_uid(fid;access_level level) -> INTEGER uid;
VFUN i_nlink(fid; access_level level) -> INTEGER links;
file_id: DESIGNATOR;


    FUNCTIONS


VFUN h_file(fid; access_level level) -> VECTOR_OF CHAR c;
    HIDDEN;
    INITIALLY
       c = ?;

OFUN f_write(fid; INTEGER offset; VECTOR_OF CHAR data;
             INTEGER uid)[access_level level] ;
    EXCEPTIONS
       ~ file_exists(fid); $( ~has write permission)
       ~ write_allowed(level, read_level(fid, level));
    EFFECTS
        'h_file(fid, read_level(fid, level))
       = VECTOR(FOR i
                  FROM 1
                  TO MAX({ LENGTH(h_file(fid,
                                       read_level(fid, level))),
                          offset + LENGTH(data) })
                : IF i < offset OR i > offset + LENGTH(data)
                   THEN h_file(fid, read_level(fid, level))[i]
                   ELSE data[i - offset]);
```

Fig. 2.1 - Specification of files (continued)

```
VFUN f_read(fid; INTEGER offset, count, uid)[access_level level]
          -> STRUCT_OF(INTEGER counted; VECTOR_OF CHAR data) rs;
    DEFINITIONS
       INTEGER file_length
          IS LENGTH(h_file(fid, read_level(fid, level)));
    EXCEPTIONS
       ~ file_exists(fid, level);
    DERIVATION
       LET VECTOR_OF CHAR v =(IF offset + count > file_length
                                 THEN
                                  VECTOR(FOR i FROM 1
                                         TO file_length - offset:
                                         h_file(fid,
                                          read_level(fid, level))
                                           [offset + i])
                                 ELSE VECTOR(FOR i FROM 1 TO count
                                        : h_file(fid,
                                          read_level(fid, level))
                                           [offset + i]))
        IN STRUCT(LENGTH(v), v);

OFUN f_trunc(fid; INTEGER uid)[access_level level];
      $( this cuts a files contents back to ?)
    EXCEPTIONS
       ~ file_exists(fid, level);
       ~ write_allowed(level, read_level(fid, level));
       ~ is_file_owner(fid, uid, level);
    EFFECTS
       h_file(fid, read_level(fid, level)) = ?;

VFUN f_length(fid)[access_level level] -> INTEGER length;
    EXCEPTIONS
       ~ file_exists(fid, level);
    DERIVATION
       LENGTH(h_file(fid, read_level(fid, level)));

END_MODULE
```

Fig. 2.1 - Specification of files (continued)

```
OFUN f_write(fid; INTEGER offset; VECTOR_OF CHAR data;
             INTEGER uid)[access_level level] ;
    EXCEPTIONS
       ? = SOME access_level l |
            read_allowed(level, l) AND i_nlink(fid, l) > 0;
       ~ write_allowed(level,
                       SOME access_level l |
                        read_allowed(level, l)
                        AND i_nlink(fid, l) > 0);
    EFFECTS
        'h_file(fid,
                SOME access_level l |
                 read_allowed(level, l)
                 AND i_nlink(fid, l) > 0)
       = VECTOR(FOR i
                  FROM 1
                  TO MAX({ LENGTH(
                            h_file(fid,
                                   SOME access_level l |
                                    read_allowed(level, l)
                                    AND
                                    i_nlink(fid, l) > 0)),
                         offset + LENGTH(data) })
                : IF i < offset OR i > offset + LENGTH(data)
                   THEN h_file(fid,
                               SOME access_level l |
                                read_allowed(level, l)
                                AND i_nlink(fid, l) > 0)[i]
                   ELSE data[i - offset]);
```

Fig. 2.2 - Expanded form of "write" operation

```
FORALL fid; INTEGER offset; VECTOR_OF CHAR data; INTEGER uid;
       access_level level:
  FORALL access_level l:
    read_allowed(level, l) => read_allowed(level, l);

FORALL fid; INTEGER offset; VECTOR_OF CHAR data; INTEGER uid;
       access_level level:
  ~(? = SOME access_level l |
         read_allowed(level, l) AND i_link(fid, l) > 0)
  => FORALL access_level l:
       (read_allowed(level, l) => read_allowed(level, l));

FORALL fid; INTEGER offset; VECTOR_OF CHAR data; INTEGER uid;
       access_level level:
  ~(? = SOME access_level l |
         read_allowed(level, l) AND i_link(fid, l) > 0)
  AND
  write_allowed(level,
                SOME access_level l |
                 read_allowed(level, l)
                 AND i_nlink(fid, l) > 0)
  => read_allowed(SOME access_level l |
                   read_allowed(level, l)
                   AND i_nlink(fid, l) > 0,
                  SOME access_level l |
                   read_allowed(level, l)
                   AND i_nlink(fid, l) > 0);
```

Fig. 2.3 - Theorems for proving security of "write"
(continued on next page)

```
FORALL fid; INTEGER offset; VECTOR_OF CHAR data; INTEGER uid;
       access_level level:
  ~(? = SOME access_level l |
         read_allowed(level, l) AND i_nlink(fid, l) > 0)
  AND
  write_allowed(level,
                SOME access_level l |
                 read_allowed(level, l)
                 AND i_nlink(fid, l) > 0)
  => FORALL access_level l:
      read_allowed(level, l)
      => read_allowed(SOME access_level l |
                       read_allowed(level, l)
                       AND i_nlink(fid, l) > 0,
                      l);

FORALL fid; INTEGER offset; VECTOR_OF CHAR data; INTEGER uid;
       access_level level:
  ~(? = SOME access_level l |
         read_allowed(level, l) AND i_link(fid, l) > 0)
  AND
  write_allowed(level,
                SOME access_level l |
                 read_allowed(level, l)
                 AND i_nlink(fid, l) > 0)
  => read_allowed(SOME access_level l |
                   read_allowed(level, l)
                   AND i_nlink(fid, l) > 0,
                  level);
```

Fig. 2.3 - Theorems for proving security of "write" (cont'd)

SECTION III
SYSTEM SECURITY STRUCTURE

The proposed KSOS system provides several different protection mechanisms and enforces several security policies. In order to understand how these mechanisms relate to one another and how they interact, it is necessary to have a clear picture of the overall structure of the system. The system has three basic parts:

1. the kernel,

2. the emulator, and

3. the program library.

The kernel provides the basic protection mechanisms and enforces the desired security policies. Since the enforcement of the security policies is dependent upon the correct design and implementation of the kernel, it is necessary that the correctness of the kernel be verified. The emulator presents the desired system interface to the users and library programs. In the case of this project, the desired system interface is to resemble that of the UNIX*tm operating system. However, other emulators could be written to provide other system interfaces, or the emulator could be entirely left out making the system interface be the kernel interface. The program library is simply a collection of utility routines to be used as the users of the system see fit. In this project the program library will be similar to the UNIX*tm system program library with the possible addition of some programs that exploit the security environment.

In choosing the security structure for KSOS, the intent has been to permit the most general and flexible interface possible that is meaningful within the context of multilevel security and that can be efficiently implemented. A more general structure, such as a general domain oriented structure, could not be efficiently implemented within the constraints of the PDP-11 architecture. The next few paragraphs explain the security provided by the kernel and emulator interfaces in greater detail.

THE KERNEL INTERFACE

The primary purpose of the kernel, from the point of view of security, is to enforce multilevel security. A formal definition of multilevel security is given in Section II of this report. This definition is called property P1. It states that information in the system may flow only upward in security level or remain at the same level. We will call multilevel security as defined by P1 strict multilevel security. The security properties P2 are actually slightly stronger and more strict than P1, however, the difference is so minor that for the purposes of this discussion they will be considered equivalent. Therefore, P1 and P2 are definitions of strict multilevel security. We would like our kernel to enforce strict multilevel security, however, there are many applications for which strict multilevel security is too restrictive, one example being the necessity to downgrade information in security level. Therefore, we find it necessary to allow the relaxation of the strict multilevel security rules under certain circumstances. The particular relaxations of strict multilevel security we have chosen were given in the previous section. All the relaxations chosen have the property that they retain some meaningful measure of security and that the security so obtained can be demonstrated to hold for the entire system by proving the correctness of the kernel. The enforcement of the various types of security of the system is illustrated by Fig. 3.1 which shows the function space of the kernel. Some of the kernel functions obey strict multilevel security, i.e., processes invoking only these functions cannot violate the properties of strict multilevel security. These functions are represented by the white area of the kernel function space. Functions in the grey areas obey less strict multilevel security, i.e., processes invoking functions in the grey areas are not subject to strict multilevel security, but still cannot violate certain less restrictive security rules. The regions of the kernel function space in which each process may operate is determined when the process is created and so it is possible to create processes that obey strict multilevel security and processes which obey other less strict security. Many applications may require process that obey only strict multilevel security and these applications will be multilevel secure in the strict sense.

There are a few kernel function invocations intended for use only by the emulator. The purpose of these special function invocations is to permit the emulator to protect its programs and data bases from user programs. These functions are represented by the darkest area of Fig. 3.1. (On a PDP-11 these special functions could be implemented efficiently by having the emulator run in supervisor mode.)

Most of the functions provided by the kernel will be implemented using a synchronous interface, i.e., the return of the invoked function to the calling program implies that the function has been completed. Some functions may be implemented by programs executing in parallel to the calling program via an asynchronous interface (e.g., IPC). In this latter case, the return of the invoked function may not imply completion of the function. The synchronous and asynchronous interfaces are separated by the dotted line in Fig. 3.1. The asynchronous interface supports security in the same manner as the synchronous interface.

THE EMULATOR INTERFACE

The emulator is constructed using the functions of the kernel. The emulator hides the special functions provided i.. the kernel for use by the emulator, but all the other functions of the kernel are potentially available to users in addition to the functions provided by the emulator. However, it will be possible to selectively restrict the access of processes to certain kernel or emulator functions. For example, certain processes may be denied access to kernel functions directly and may only have access to emulator functions; certain processes may be permitted access only to the functions of the kernel that enforce strict multilevel security, thereby assuring that the process operates in a strict multilevel secure fashion. The function space as seen by a user process is depicted in Fig. 3.2. In this figure the darkest region represents the functions implemented by the emulator. Access by each process to any of the regions in the function space may be restricted at the time of process creation. It is possible that, within a process, the emulator has access to regions in the kernel function space that the user programs do not. This feature makes it possible to have the emulator enforce some security policy that the kernel does not enforce, such as a policy that calls for the downgrading of the security level of information but only under certain special circumstances.

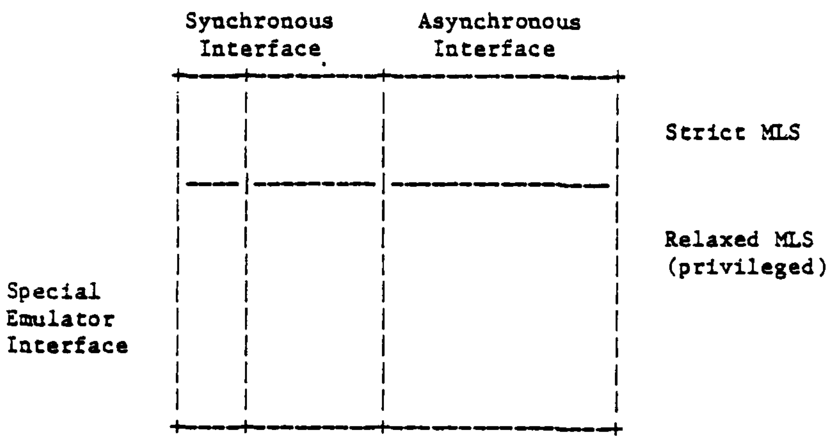
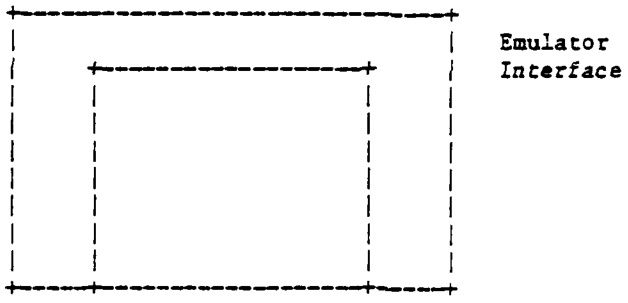


Figure 3.1 - Kernel Function Space



Function 3.2 - Emulator Function Space

SECTION IV
LANGUAGE SELECTION FOR KSOS IMPLEMENTATION

This section considers the task of selecting a programming language for the proposed implementation of KSOS. Desired characteristics of an appropriate language are identified. In fact, these characteristics are essentially a subset of those of the [July 1977 revised] IRONMAN specifications, although generally some accommodation of the SRI methodology is required. Euclid, Modula, Pascal, UCLA Pascal, Concurrent Pascal, Gypsy, C, and ILPL are considered here. None of these languages currently meets all of the desired requirements. However, Euclid and Modula appear to be particularly strong candidates, the former subject to the suitability of the implementation currently in progress, and the latter subject to support of certain language extensions currently in progress.

The main conclusion here is that selection of the implementation language need not be made during Phase I, and can safely be put off until about 1 July 1978. Instead of choosing a language at this time, a decision procedure is given as to how that selection should be made. This procedure is based on the analysis of this section, but will use knowledge expected to be available in July. There are several reasons why such an approach is appropriate. First, no substantial amount of code (except for preliminary exploration) will be written until well into Phase II. Second, more will be known around July about the current Euclid compiler development and about the current effort to extend the Modula language and its compiler. Both of these efforts are scheduled to be sufficiently well along by then. Third, there appears to be relatively little effort required before July by the KSOS implementation team in order to assure the availability of suitable support for the chosen language. (However, it may eventually be desirable to develop a profiler-debugger.) Thus, overall, the impact of deferring the language choice remains negligible, at least until about July.

OVERVIEW

Table 4.1 summarizes the requirements for a KSOS programming language (KPL) suitable for implementing the KSOS kernel and UNIX*tm emulator. These requirements are similar to a subset of the IRONMAN requirements, as seen from a summary comparison of the two sets of requirements given in Table 4.2.

Various programming languages are compared. Table 4.3 provides a comparative summary of how well each language satisfies the various requirements of Table 4.1 at present, and also indicates anticipated improvements. It is seen that Euclid and Modula are the strongest candidates when projected to the time at which the programming language is needed, with Euclid having a distinct edge based on its current development status. This section then concludes with some KSOS implementation considerations, illustrated by the sketch of a Euclid program for the example module of Section II (Figure 4.2).

REQUIREMENTS

The requirements for the KSOS programming language are summarized in Table 4.1. These requirements contribute in various ways to the intrinsic security of the system, the intrinsic correctness of the implementation, the ease of coding, the understandability of the programs, and the ease of program verification. It should be noted that many of these requirements are in fact highly beneficial, but not in each case mandatory. However, the additional effort necessitated by their absence is in some cases considerable.

It is difficult to rank the requirements in order of decreasing importance, since their consequences are diverse. However, some of the motivations for these requirements are noted as follows.

* The language must be well supported by an effective compiler capable of producing efficient code for the DEC PDP-11/70. Without such support, all other requirements are meaningless.

* Program-defined data types are desired in order to support data abstraction for virtual resources. Encapsulation of these data types is desirable to avoid a significant and pervasive type of security flaw (see below), and to increase provability.

* Dynamic creation and deletion of objects of program-defined types is highly desirable, although not essential. Dynamic creatability considerably increases the intrinsic safety of implementation, for example by sharply reducing the likelihood of data residues due to incomplete deallocation, by enforcing encapsulation of the code that performs allocation and deallocation of virtual resources. It also increases the readability of code, and simplifies the verification effort.

* The language should be strongly typed and type-safe. However, a controllable facility for explicit type conversion or for union types is necessary, for example, when treating a stack of mixed-type elements.

* Multiprogramming must be supported --at least in some simple form.

* The writing of machine-dependent pieces of code must be supported, particularly to manage input-output and secondary storage devices.

* Separate compilation is operationally highly desirable. As a simple example, the operating system must be compilable separately from user programs. Also, the kernel, the non-kernel security-related software, and the UNIX*tm emulator should be separately compilable. Further, during development, it is very useful if pieces of the kernel or of the emulator can be separately compiled, even if in a production version

they are not. (The compiler should be able to compile the system as a unit as well as in separate pieces.)

* Finally, although it is not an immediate concern of the KSOS effort, the language should do very little that would hinder formal verification of the consistency between programs and formal specifications. In fact, whether or not such verification is ever attempted, constraining a language such that its code could be reasonably verified can yield many intrinsic benefits in terms of the reliability, understandability, and maintainability of the resulting system.

A somewhat orthogonal view of the requirements for a KPL is motivated by some pragmatic considerations of existing (insecure) systems. Bisbey and his colleagues at ISI have catalogued and studied various characteristic security flaws that in the past have plagued systems attempting to be secure. (See Bisbey et al. [75], Bisbey et al. [76], Carlstedt [76], Carlstedt et al. [75], Hollingworth and Bisbey [76].) They have focused on three classes of flaws, namely the inconsistency of data over time, the nonvalidation of critical conditions and operands, and residues resulting from incomplete deallocation. They have also identified other categories, namely problems associated with serialization, interrupted atomic operations, exposed representations (noted above), aliasing, incorrect domain usage, and incorrect operation selection. In general, through a combination of good methodology, good language design, good compiler diagnostics, and appropriate restrictions on language use, many of these flaws can be categorically avoided.

An analysis of how the use of the SRI methodology and the appropriate choice of programming languages can contribute to the avoidance of these characteristic flaws is found in Neumann [78]. The methodology itself contributes to the avoidance of most of these flaws with respect to the design, by constraining the way in which specifications are written (without constraining what may be specified.) The suitable choice of programming language can have similar impact on the implementation, by constraining the way in which programs must be written. For example, data inconsistency of parameters cannot arise in specifications; in implementation, it can be avoided by requiring call-by-value. Exposed representations are avoided in the design by the specification language, and further avoided by suitable language support for data abstraction. Nonvalidation is largely avoided by strong type checking, explicit subtypes (e.g., ranged variables), and explicit exception conditions. Residues cannot arise in well-formed specifications, and can be avoided in implementation by encapsulation of simple algorithms for deallocation. However, some of the responsibility must be handled by the language, the compiler, and associated language support tools.

COMPARISON OF THE CANDIDATE LANGUAGES

The languages considered in this study as candidates for the KPL are summarized in Table 4.3. The letter grades given in the table are obviously subjective. However, they have been subjected to the scrutiny of various knowledgeable language people to check their relative integrity.

The languages considered here include a variety of Pascal-based languages, Euclid, Modula, UCLA Pascal, Gypsy, and Pascal itself. Concurrent PASCAL was also considered, but seems similar enough to PASCAL except for its handling of multiprogramming to not require independent treatment. In addition, it is designed to rely on the existence of an operating system for some of its features, and thus is not appropriate for implementing an operating system. For practical completeness, the main UNIX*tm language, C, is also considered --although it presents numerous problems as a would-be KPL, especially if program verification is ever to be considered. In addition, a language designed to be ideally suited to the SPI methodology is also included, although it is as yet not supported. That language is ILPL, discussed in the Ford/SRI proposal, and documented in Neumann et al. [77]. (The Xerox PARC language MESA might be a strong candidate, were it not for the fact that it is nonexportable.) Finally, in order to enhance assessment of future alternatives, the IRONMAN requirements are represented in the form of an as yet undefined language that would satisfy those requirements.

All of the candidate languages are fairly precisely defined. However, there are difficulties when it comes to finding a well supported language that also has the desired features. Euclid is seen to be quite reasonable in this respect. A Euclid to C transliterator now exists, and a compiler (translating to PDP-11 code) is expected to be available around 1 July 1978, in an effort at the University of Toronto and I.P. Sharp. Modula exists in a 6400 version that is not yet widely available, along with a PDP-11 version, at the Eidgenossene Technische Hochschule in Zurich. Recent discussions with Niklaus Wirth indicate that most of the difficulties in using the current version of Modula for KSOS are being surmounted by changes already contemplated by Wirth in Zurich, and that a viable compiler supporting most of these changes is planned for around 1 July 1978. (Another recent Modula implementation also exists on a PDP-11 at York University in England.) Various Pascal implementations exist. UCLA Pascal is being used in the UCLA security kernel, and is currently undergoing both language extensions and improvements in efficiency. Gypsy is supported only in terms of the front-end checking of the Texas verification system, although the extension of that support into a more compiler-like environment is in progress. ILPL is at present unsupported, although support is planned.

C. A. R. Hoare has suggested that no good language can be developed in less than ten years. Modula, as one language in a series of Pascal-like languages developed by Niklaus Wirth, is probably the most seasoned of the candidate languages. Euclid might appear weak in this category, but it too is conceived incrementally to Pascal, and its design represents considerable experience. Gypsy is also fairly strong in this respect. ILPL would appear to be the least seasoned, but compensates by being the simplest of the candidate languages --because of its relationship with the surrounding Hierarchical Development Methodology. HDM provides many features traditionally a part of the programming language, such as the data structures at each level out of which higher levels are implemented. As a result, it has the potential of greatly simplified verification: many of the proofs arise from properties of the methodology or provable properties of the specifications, rather than from properties of the programs.

Modula, Pascal, and UCLA Pascal are deficient in their lack of support for separate compilation. Most of the languages except Gypsy and ILPL are deficient in their handling of exception conditions. Support at least for error conditions is certainly required; ILPL provides more, with a general exception-handling mechanism well suited to the formal specifications of the SRI methodology.

The control features as well as the data types and data structures supported by each of the candidate languages are more or less satisfactory. C is deficient in its almost total lack of strong typing, and most of the languages (except Euclid, Modula, and ILPL) are deficient in their handling of data abstraction and encapsulation for program-created data types. Dynamic object creation is deficient or lacking in C, Modula (at present), and Gypsy (at present).

The handling of multiprogramming and the handling of machine dependence both present fundamental differences of viewpoint. One alternative is to leave the handling to language extensions, and to have the language do essentially nothing (neither favorable nor unfavorable). A second alternative is to provide some mechanism within the language. This approach may be appropriate for some applications, but inappropriate for others, as in the case of Gypsy and to some extent Modula.

A major requirement of the selected programming language is that it be compatible with the SRI hierarchical development methodology. Here of course ILPL has an advantage, since it has been developed in conjunction with the methodology. The only language that really is bad in this respect is C, although each of the others presents some problems. Euclid seems reasonably appropriate.

A further consideration is verifiability. Here Euclid, Gypsy and ILPL seem superior, largely because provability was a major consideration in each language design. (The future role of program verification is considered in Appendix IV.A.)

DECISION PROCEDURE FOR LANGUAGE SELECTION

Based on the above evaluation, the following procedure is recommended for selection of the KSOS programming language, on or about 1 July 1978.

```
IF the Euclid compiler development effort at Toronto con-
   tinues on its present course and gives adequate support,
   THEN use it -- with some accommodation for hardware
   error signalling. Some subsetting of the language
   may be desirable, for simplicity, efficiency, and
   enhancement of any eventual verification.
ELSE IF Modula has been adequately extended and those
   extensions supported, THEN use it.

   {At present it seems that Euclid will be appropriate,
   although Modula provides an attractive alternative.
   At this point, it is extremely unlikely that any more
   ELSE clauses are needed. However, there are other
   languages that might be appropriate, e.g.,}

ELSE IF ILPL is adequately supported, THEN use it.
ELSE IF Gypsy is adequately supported, THEN use it.
ELSE use UCLA Pascal.

   {It is remotely feasible to use C, but only with the
   addition of various programming constraints, and with
   the explicit understanding that the kernel would
   subsequently be recoded in a more suitable language,
   before any program verification is attempted.
   However, such a course is neither necessary nor
   advantageous.}
```

IMPLICATIONS OF HAVING MADE THE CHOICE OF LANGUAGE

The following sections present some of the considerations arising once the language choice has been made. It is of course important to assure that these considerations are anticipated in the language choice itself.

LANGUAGE-INDEPENDENT IMPLEMENTATION CONSIDERATIONS

Various considerations arise in implementation that are largely independent of the specific language choice. Some of these are summarized here.

(1) The user-emulator and emulator-kernel boundaries represent specific interfaces at particular levels in the hierarchical specifications, but are not explicitly designated as such in the specifications. The same is true of the security perimeter, namely the interface which includes all security-relevant code (namely the kernel and the trusted processes, i.e., non-kernel security-related code). These boundaries are made explicit by the abstract machine interpreter for each level, which knows whether that level is

part of user mode, supervisor mode, or kernel mode. Thus these decisions become a part of the initialization process.

(2) Specifications and mapping functions are language-independent in spirit, although the data structures of a particular language and of the target machine (in this case, the PDP-11) will influence the way in which mapping functions are written at the lowest levels.

LANGUAGE-DEPENDENT IMPLEMENTATION CONSIDERATIONS

Numerous considerations arise that are dependent on the language choice. Four are identified here, relating to the transformation of SPECIAL constructs into the programming language, the handling of certain implementation concepts not addressed by the specifications, the generation and optimization of programs, and program verification.

(1) Transformation of SPECIAL constructs into the chosen programming language, involving the following:

* Designators, i.e., protected names for abstract objects (e.g., represented as nonmodifiable integers)
* Modules, including the facilitation of encapsulation and module integrity (e.g., by establishment of suitable constraints on import/export or their equivalent)
* Types and type checking, including assurance of type safety
* Structures of the various levels of data abstraction and assurance of data structure integrity
* Facilitation of proper synchronization among potentially parallel operations by establishment of synchronization conventions (note that the specification of each function is logically indivisible),
* Exception conditions: associating names with exceptions (Note: SPECIAL now supports named exceptions directly.)

(2) Abstract machine interpreter issues not visible in the specifications, but significant to the implementation. The abstract machine interpreter deals with the wav in which the functions at each level are executed, and deals with function invocation, exception handling, and seouencing:

* Function invocation options: software expansion vs. hardware expansion/interpretation/execution. The former includes a choice among in-line macro expansion, procedure calls, process invocations via IPC (even intersystem invocations via network protocols).

* Calling conventions: argument and return value passing, normally permitted on call-by-value and return-by-value basis only; deviations (e.g., pointer passing or conversion to call-by-reference, for efficiency reasons) should be explicitly sanctioned on an individual-case basis.

* Exception handling: signalling exceptions, treating hardware

errors as a special case or as an instance of the general case. Note a desired symmetry between return values and exception returns.

(3) Program generation and optimization

* Transformation of EFFECTS and of EXCEPTION checking into abstract programs
* The selective omission of code for exception detection and exception handling, where avoidable --e.g., because of compiler checks, loader checks, or verification-guaranteed properties
* Handling of remaining exceptions
* Handling of explicit optimization instructions and other optimization
* Transformation of abstract programs into executable code

(4) Program verification

* Assurance of general consistency with specifications, or assurance of partial consistency such as proper placement of synchronization primitives, lack of storage residues, etc.
* Support for the chosen language in the front-end of the verification environment

ILLUSTRATION OF A EUCLID PROGRAM

As an example of how these concepts apply, the module specification of Figure 2.1 of this report is taken as the basis for an illustrative implementation. Mapping functions relating this module to lower-level modules (not specified here) are sketched in Figure 4.1. (In general, the mapping functions provide the basis for the assignment of explicit data structures preparatory to implementation. However, the lower levels are not essential to the illustration, and thus are omitted.) Finally, a Euclid program for the fwrite function of that module is sketched in Figure 4.2. (Again, since the lower-level specifications are omitted, the program details relating to the specific implementation are omitted.)

Various relationships between specifications and Euclid programs may be observed from this illustration. These include the similar roles of modules used for encapsulation of both procedure and data abstractions in SPECIAL and in Euclid; the similar roles of procedures in each case; the similarities between respective type and data declarations, and strong type checking (although the repeated writing out of type information is avoided by the more implicit style of Euclid). Import lists for modules and procedures in Euclid directly reflect several SPECIAL constructs, namely external references from lower-levels modules, module parameters, exception conditions and return values to be returned, and function arguments. Export lists in Euclid modules directly reflect the list of functions visible at the level in which the specified module appears. These correspondences should make it quite straightforward to program in a language such as Euclid, given specifications in SPECIAL, and should aid in verifying consistency of programs and specifications.

These correspondences would also make possible an elementary kind of automatic programming, in which a program skeleton (similar to Figure 4.2) is generated automatically from specifications --presumably after those specifications have been proved consistent with the formal requirements. In this way it would be possible to avoid a large number of routine programming errors, and to stylize the form of the resulting programs for increased readability. Constructs of the specifications that could not be directly transformed into constructs of the programming language could be flagged, and perhaps indicated as comments in the program skeleton.

Although many SPECIAL constructs transform directly into corresponding programming language constructs, some do not. For example, an exception condition in a specification normally corresponds to an explicit check in the program. However, in certain cases (e.g., if the exception is implicit in an EXCEPTIONS OF construct), the exception condition might be transformed into a Euclid "assert" statement (whence it is to be shown that it cannot occur). (On the other hand, a SPECIAL "assert" statement would always be transformed into a Euclid "assert" statement, appearing as a precondition.) A programming tool that would provide skeletal programs wherever that would be appropriate is in fact being contemplated for development under other contracts at SRI, and has a good chance of being ready in time for use in Phase II of KSOS. However, any such tool must be used with discretion by the programmer, for a specification is not meant to dictate its implementation; the specification is intended to provide clarity of understanding, whereas the program must be written for efficient implementation. Nevertheless, such a transformation into a skeletal program with indications of what remains to be done seems to be simple to achieve and useful in reducing programming error.

CONCLUSIONS

The final selection of the KPL can safely be deferred until about 1 July 1978 (Phase II), at which time much more will be known about the candidate languages. At present, based on expectations of support anticipated at that time, the use of Euclid seems most desirable, although possibly with some restrictions on the use of the language. Progress on the existing Euclid compiler development at Toronto and support by I.P.Sharp both seem to be indicative of the timely delivery of a compiler capable of producing reasonably efficient code for the DEC PDP-11/70.

Modula might alternatively be used -- assuming success of the current modifications to the language and the compiler. However, the Modula support picture is at present much less clear. Gypsy and ILPL are both interesting languages, but could not be considered as serious candidates unless they are adequately supported at the time the decision must ultimately be made. UCLA Pascal could be appropriate if Euclid and Modula fail. Pascal and Concurrent Pascal have enough disadvantages to be not worth considering further at this time. C would be suitable only with extensive effort, not considered necessary or appropriate for KSOS.

It should be noted that the DoD/1 effort has led to the definition of four candidate languages, with varying degrees of suitability for KSOS. Although none of these DoD languages is likely to be supported in time for the Phase II implementation of KSOS, the language(s) ultimately emerging from this effort could still play a role in subsequent versions of KSOS (See Appendix IV.A).

APPENDIX IV.A
THE POTENTIAL ROLES OF PROGRAM VERIFICATION AND NEW LANGUAGES

VERIFICATION AND NEW LANGUAGES

Verification is at this point is an important but not overriding concern in the language evaluation. It appears that Euclid is suitable for eventual program verification, particularly with the imposition of a few language restrictions. It is felt that with such constraints on the chosen language, the KSOS kernel programs can be formally verified using either Euclid or Modula.

A potential future role for DoD/1 is possible. If at some later time an effort is to be mounted to verify the programs of the KSOS kernel and trusted processes, it may then be appropriate to reevaluate the available languages, considering at least DoD/1, ILPL, and Gypsy as candidates as well as Euclid and Modula -- for different reasons. The choice of language at that point should then include suitability for verification as a major criterion. Reimplementation of the kernel and trusted NKSR programs would seem quite straightforward, considering their fairly small total size.

DOD/1

If DoD/1 were later chosen, it would then be relatively easy to recode the KSOS kernel. This would be enhanced since the resulting DoD/1 language would be a Pascal-based language (and is likely to resemble both Modula and Euclid in various essential respects). This would also be enhanced by the Ford/SRI design, since formal specifications for the resulting kernel will exist for which the basic multilevel security properties have been proved -- with respect to the design. These specifications would be essentially the same for both the prototype unverified implementation and any subsequent verified implementation.

ILPL

Although ILPL is not supported sufficiently to warrant its use at present, there are two potential roles that it might take in the future, both outside of the scope of Phase II. The first involves its use as an intermediate language in a subsequent program verification, serving to simplify the verification effort. The second involves its use as an abstract programming language in which to export abstract programs for the reimplementation of the design on other hardware. At present, these roles remain speculative.

The recoding of the security-related software of KSOS in ILPL at some subsequent time when program verification is to be seriously attempted would have several major advantages -- assuming ILPL is by then supported as planned. ILPL provides an abstract programming language, in the sense that one level in the system hierarchy can be programmed using just ILPL constructs and the data abstractions and functions provided by the specifications of the next lower level(s) in the hierarchy. Thus the data abstractions used by ILPL for implementing a particular level of the design are those supported by the hardware and those that are defined by lower-level specifications. Consequently, the ILPL programs may be (automatically) translated into whatever target language is desired --e.g., Euclid, DoD/1, or direct to machine code-- while the program verification can be conducted by relating the specification language SPECIAL to ILPL rather than by having to work directly with the more complicated target language. In addition, the translation to the target language may be verified in a largely program-independent (but language-dependent) manner. As a result, the verification effort would then be decomposed into model-to-spec consistency proofs, spec-to-ILPL consistency proofs, and generic treatment of the ILPL to target language step. This approach is considered to be considerably simpler than the traditional program verification approaches of relating verification directly to the target language, and holds significant promise for the future of system verification.

ILPL could also be useful as a reference language, in that an ILPL abstract-program version of the KSOS kernel, together with the formal specifications, could increase the ease of implementation on other machines (e.g., the Honeywell Level 6/40) or on the same machine with other programming languages. Exportation of such abstract programs would further enhance the compatibility between different implementations of the KSOS kernel, and increase the understandability of the system.

INCREMENTAL VERIFICATION

Associated with the notion of verifying a system design and its implementation are various related notions dealing with reverification of subsequent modifications, and with verification of different implementations of the same specifications on different hardware. Whenever changes occur in the specifications, it is necessary to check whether those changes affect the proofs of requirements. Similarly, changes in implementation or different implementations require corresponding reverification or new verification of program consistency. The SRI methodology tends to minimize such subsequent efforts.

REFERENCES

Bisbey et al. [75] R. Bisbey II, G. Popek, J. Carlstedt, Protection Errors in Operating Systems: Inconsistency of a Single Data Value Over Time, ISI/SR-75-4 (December 1975).

Bisbey et al. [76] R. Bisbey II, J. Carlstedt, D. Chase, Data Dependency Analysis, ISI/RR-76-45 (February 1976).

Carlstedt [76] J. Carlstedt, Protection Errors in Operating Systems: Validation of Critical Conditions, ISI/SR-76-5 (May 1976).

Carlstedt et al. [75] J. Carlstedt, R. Bisbey II, G. Popek, Pattern-Directed Protection Evaluation, ISI/SR-75-31 (June 1975).

Hollingworth and Bisbey [76] D. Hollingworth and R. Bisbey II, Protection Errors in Operating Systems: Allocation/Deallocation Residuals (June 1976).

Neumann et al. [77] Peter G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, A Provably Secure Operating System: The System, Its Applications, and Proofs, SRI Final Report, Project 4332 (11 February 1977).

Neumann [78] Peter G. Neumann, Computer System Security Evaluation, <u>Proc. National Computing Conference</u>, Anaheim CA (5-8 June 1978).

Table 4.1
Summary of KSOS Programming Language Requirements

Language well defined
Adequately supported by compiler and other tools
Efficient in compilation and (particularly) in execution

Control:	Data types:	Data structures:
IF ... THEN	INTEGER	ARRAY
CASE	BOOLEAN	STRUCTURE
WHILE ... DO	CHAR STRING	
PROCEDURE	BIT STRING	
FUNCTIONS	REFERENCE	
EXCEPTIONS	ENCAPSULATION for program-defined types	

Type safety
Support for separately compilable programs
Dynamic creation and deletion of objects
Support for multiprogramming
Support for machine dependence

Compatibility with the SRI hierarchical methodology
Suitability for eventual program verification

Help in avoiding characteristic security flaws
(cf. Bisbey; see Neumann [78]) such as:
- improper domain choice
- exposed representations
- data inconsistency
- naming problems
- residues
- nonvalidation
- improper indivisibility
- improper serialization
- wrong operation choice

Table 4.2

KSOS Programming Language Desiderata Tabulated
According to July 1977 Revised IRONMAN Requirements

---

1. General Design Criteria: No essential differences (NED). (Generality, Reliability, Maintainability, Efficiency, Simplicity, Implementability, Machine Independence, Formal Definition)

2. General syntax: NED.

3. Types: NED, including definition of new data types (3C) and operations between types (3-5D). However, 3-1A, floating point is not necessary.

4. Expressions: NED.

5. Constants, Variables and Declarations: NED.

6. Control Structures: NED.

7. Functions and Procedures: NED.

8. Input-Output Facilities: NED.

9. Parallel Processing: NED.

10. Exception Handling: NED, although suppressing of exceptions (10G) is not recommended unless an exception has been proved not to occur.

11. Specifications of Object Representation: NED.

12. Library, Separate Compilation, and Generic Definitions: NED.

13. Support for the Language: NED.

Table 4.3
Summary of Candidate Languages in Terms of
Suitability for Implementing KSOS

Languages	C	EUCLID	MODULA	UCLA	PASCAL	GYPSY	ILPL	DoD/1
Features				PASCAL				(4)
Support	A	(1)	C/y	B	A	(2)	(3)	(4)
Efficiency	A	(1)	A	B-/y	B	(2)	(3)	(4)
Readability	B	A/r	A	A	A	A	A	A
Ease of coding	B	A/r	A	A	A	A	A	A to B
Control	A	A	A	B	B	A	A	A
Sep Compilation	A	A	C/y	C/x	C/x	A	A	A
Arg passing	C/a	A	A	A	A	A	A	A
Exceptions	C	C	B	C	C	A	A	A
Data types	B-	A	A/py	B	B	A/p	A/p	A
Data structures	A	A	A	A	A	A	A	A
Strong typing	F	A	A/g	B	B	A	A	A
Data abstract'n	F	A	B/y	B	F	A	A/m	A
Encapsulation	F	A/q	A	B	D	B/q	A	A
Dyn creation	D	A	D/y	A	A	D/y	A	A
Multiprogram'ng	UNIX	Extend	B	Extend	Extend	B	Extend	A to C
Device-depen'ce	A	A	B	Extend	Extend	Fy	Extend	A(Ext)
HDM suitability	F	A-	B-	C	C	B+	A	A-
Verifiability	F/c	A/r	B	B	B	A	A	A to B

( ): Work planned. Evaluation incomplete or premature.
1: EUCLID to C transliterator works now. EUCLID to 11 code planned July 78. A generic caveat applies to those entries in the EUCLID column for which EUCLID is rated better than PASCAL or MODULA: difficulties in implementation could lead to a desire for language changes; however, based on experience to date, this does not seem likely.
2: Presently only syntactic and semantic analysis exists as part of a verification environment. Implementation and language redesign in progress.
3. ILPL is at present only a paper language -- although simple and complete.
4. The four candidate DoD/1 languages have been defined, each claiming compliance with the IRONMAN requirements. The entries given for the DoD/1 column indicate the variety provided by these languages. A compiler for at least one language is expected to result from subsequent work.

a: In the UNIX code, arguments are immediately copied following each call.
c: Supersubset used with preprocessor could lead to verifiability, although probably with significant loss of efficiency.
g: No generic types.
m: In combination with the SRI methodology, which provides data abstraction in the specifications for each level of a hierarchical design.
p: No pointers. In ILPL, designators seem to be an acceptable substitute.
q: Encapsulation can be achieved by appropriate language constraints (e.g., on import/export), but is not intrinsic in the language.
r: With a few restrictions on the use of the full language.
x: Not currently supported.
y: Improvements planned in a version of the language under development.
Extend: Language design intends this to be implemented via extensions.

Figure 4.1
Skeletal Mapping Functions for the Files Module of Fig. 2.1

```
MAP Files TO Metafiles, Machine

     TYPES $(See Fig. 2.1.)

     DECLARATIONS $(See Fig. 2.1.)

     PARAMETERS $(See Fig. 2.1.)

     DEFINITIONS $(See Fig. 2.1.)

     EXTERNALREFS

     FROM files:
VFUN h_file(file_id fid; access_level level) -> VECTOR_OF CHAR c;

     FROM metafiles:
VFUN i_uid(file_id fid; access_level level) -> INTEGER uid;
VFUN i_nlink(file_id fid; access_level level) -> INTEGER links;
file_id: DESIGNATOR;

     FROM machine:
char: { VECTOR_OF boolean v | LENGTH (v) = 8 };  $(PDP-11 byte)

     MAPPINGS
h_file (file_id fid, level): ... $(VECTOR_OF char in terms of machine
                             words not necessary for purposes of the example)
END_MAP
```

Figure 4.2
Skeleton of Euclid Program for the "Files"
Module Specified in Fig. 2.1

```
   ...
{declarations in outer scope}
      var clearance : integer 0 .. maxClearance;
      var categorySet : array 0 .. numberOfCategories - 1 of Boolean;
      type accessLevel = record
         securityClearance clearance
         securityCategory categorySet
         integrityClearance clearance
         integrityCategory categorySet
      var fid : integer;
      var offset : integer;
      type data = array ... of char;
      var uid: integer;
      var [level] : accessLevel; {implicit argument, not provided
                                  by calling program}
   ...


var Files:
   module

      imports (var Metafiles, var Machine,        {lower level modules}
               var errorNo,                    {exception code return arg
               maxClearance, numberOfCategories,   {module parameters}
               eNoError,eNoFile,eBadLevelForWrite, {exception conditions}
               fid, offset, data, uid, [level];     {function arguments}

      exports (FRead, FWrite, FTruncate);          {visible functions}

         {This module includes the visible procedures FRread, FWrite,
         and FTruncate, and the hidden V-function HFile, as specified
         in Fig. 2.1 of this report.  For simplicity, only a portion of
         the code supporting the visible function FWrite is shown here.
         The representation for HFile would be found in the mapping
         functions for the Files module (Figure 4.1), were they complete.
         The argument "[level]" is imported to the module as an implicit
         argument.  That is, it is  not explicitly presented by the
         calling program, but is provided upon each call (by an abstract
         machine interpreter function).
         Note: As in Fig. 2.1., a would-be error value eReadNotAllowed
         is suppressed   favor of the error value eNoFile, to prevent
         a leakage channel.}

      const eNoError := 0;
      const eNoFile := 1;
      const eBadLevelForWrite := 2;
```

(Figure 4.2, continued)

```
      procedure FWrite =

         imports (var Metafiles, Machine,
                  fid, offset, data, uid, level,
                  var errorNo, eNoFile, eBadLevelForWrite);

         pre ();   {none; note EXCEPTIONS in the specifications are
               explicitly programmed into the begin .. end.}
         post (... {if eNoError then EFFECTS of specifications,
                if eNoFile or eBadLevelForWrite then no EFFECTS});

         begin
            begin
               exit when ... {eNoFile};
               exit when ... {eBadLevelForWrite};
               for ... {each char} loop
               ... {perform write for each character}
               end loop;
            end;

            if ... {eNoFile} then
               errorNo := eNoFile;
            if ... {eBadLevelForWrite} then
               errorNo := eBadLevelForWrite;
            else errorNo := eNoError;

         end FWrite;

      ...

   end module; {Files}
```

SECTION V
TOOLS SUPPORTING THE KSOS DEVELOPMENT

In order to facilitate the use of the hierarchical development methodology, various on-line tools have been developed or are planned. This section describes these tools, whose use is being pursued or anticipated to support the design and implementation of KSOS, as well as the way in which the verification of the design and the implementation will be pursued.

TOOLS TO SUPPORT THE DESIGN AND THE CORRESPONDENCE PROOFS

SRI International has developed an on-line environment to support the first four stages of the methodology, i.e., the interface definition, the hierarchical decomposition, the specifications, and the mapping functions. This environment also forms the basis for performing the proofs of correspondence between the formally stated multilevel security requirements and the specifications. The environment is open-ended, and -- based on existing tools -- is expected to be extended to support verification of the desired design properties and to support implementations and proofs of implementations.

The environment currently runs on TOPS-20 at SRI, existing in three parts, as follows.

(P1) The HIERARCHY MANAGER, which permits the establishment of a hierarchy of collections of modules, and which is responsible for maintaining the design structure.

(P2) The SPECIFICATION ANALYZER, which determines if each module specification is syntactically correct. This part includes type checking.

(P3) The MAPPING FUNCTION ANALYZER, which determines if the mapping function expressions are syntactically correct and syntactically consistent with the specifications of the modules involved.

These tools have proved to be very useful in the detection and correction of design errors, first in the formal specifications for UNIX*tm, and then in the formal specifications for the KSOS kernel. Many of these design mistakes are typically of the kind that would otherwise persist into the implementation.

Given formal specifications for the kernel, expected to satisfy the multilevel security properties (with a few well-documented exceptions needed to support the NKSR software), proofs that those specifications actually satisfy multilevel security are relatively straightforward, although somewha tedious. (See Section II of this report.) These proofs could well be carried out by hand. However, much of the mechanism for doing these proofs automatically already exists. Thus, a fourth tool is both feasible and highly desirable, in order to perform the correspondence proofs automatically.

(P4) The MODEL CONSISTENCY CHECKER, which performs the syntactic checks for correspondence proofs that are not a part of the specification language syntax checking, which performs simple semantic checks, and which also generates logical formulae whose validity is equivalent to the satisfaction of the more complicated semantic conditions for consistency with the model. The basis for this checker is summarized in Section II of this report. It is expected that the deductive system of the Boyer-Moore verifier developed at SRI International will be directly usable.

Based on experience to date, the generation of the logical formulae is straightforward. Doing proofs automatically will be helpful in eliminating human error from the proof process. Essentially all of the correspondence proof effort can be mechanized by these tools. That is, all but a few special cases of demonstrating that the specifications satisfy the required multilevel security properties are syntactic in nature, and can be treated automatically. The remaining cases, once identified, can be characterized, and most of those can then be treated automatically from then on by generalizing the special cases.

TOOLS TO SUPPORT IMPLEMENTATION AND PROGRAM VERIFICATION

In addition to the tools outlined above to support the design and the correspondence proofs, various related tools are planned or actively being developed at SRI to support implementation and program verification in general. These additional tools are being developed under other contracts, so that there is no expected cost to the KSOS effort, other than the acquisition of these tools and the accommodation of the KSOS Programming Language.

(P5) The PROGRAM HANDLER, which determines if each program is syntactically correct. This tool may also be extended to perform simple semantic checks on the programs, such as those for the placement of synchronization primitives to assure nonharmful modification of shared state information.

(P6) The DEVELOPMENT DATA-BASE MANAGER, which maintains a data base of the specifications, programs, and proofs in (P1), (P2), (P3), (P4), and (P5), keeping track of which modules are specified, mapped, implemented, and verified.

(P7) Additional proof tools to support proofs of consistency between specifications and programs. If program verification is ultimately to be undertaken, these tools would include

(a) A parser for the KSOS programming language (KPL), possibly the parser of the KPL compiler itself.

(b) A verification condition generator for the KPL. This VCGEN would generate verification conditions whose correctness is necessary to guarantee the correctness of the system implementation.

(c) A translation mechanism from SPECIAL to existing SRI verification tools.

(d) The adaptation of a program prover suitable for proving the verification conditions generated in (b) above. It is currently expected that the existing SRI theorem prover of Boyer and Moore would be adapted with relatively little effort, and used together with an existing SRI formula simplifier due to Shostak (the latter providing a decision procedure based on logic and arithmetic expressions). The RADC proving environment is also expected to play a role, probably through the merging of its user interface with the Boyer-Moore verifier. Both the RADC work and the Boyer-Moore work (supported by NSF) are strongly funded for the relevant future. On the basis of existing work, a viable proving environment is expected to exist at the time it would be needed for KSOS. In addition, the Good, London, and Luckham verifiers could also provide competitive tools that might also be incorporated if they were deemed appropriate. However, these other verifiers at present present various difficulties concerning their adaptability to the SRI methodology and to hierarchically structured programs, and the applicability to the chosen KPL.

A usable collection of these tools is expected to be available sufficiently early in Phase II to be useful for carrying out illustrative proofs of program correctness.

AUTOMATING THE DESIGN PROOFS

The technique for proving the consistency of the KSOS design with the security model has already been described in Section II. The approach to automating the proof is indicated in (P4) above. For the purposes of automation the proof of security of the design can be divided into three tasks:

(T1) determining that the design specifications are well formed (i.e., that the syntax is correct and that the strong typing is enforced),

(T2) generating the theorems from the specifications that must be proved to assure security, and

(T3) proving these theorems.

As described above, SRI has developed a tool (P2) that accomplishes task (T1). The theorem-proving tools mentioned in (P7) above can be used for accomplishing task (T3). The theorem prover developed by Boyer and Moore seems aprropriate, because its operation is largely automatic and requires minimal user interaction. (The fact that the theorem prover deals with LISP is really incidental, in that LISP is used primarily as an internal form for recursive programs.) Thus, the only tool that need be written is the theorem generator for accomplishing task (T2). This theorem generator can take the internal parsed form of a specification generated by the specification checker and produce theorems suitable as input to the Boyer-Moore theorem prover. FACC estimates that programming this theorem generator would require one and one-half man-months of effort for an experienced LISP programmer having some familiarity with the Boyer-Moore theorem prover and the specification checker.

AUTOMATING THE IMPLEMENTATION PROOFS

Informal checks that the high-level language implementation programs and the formal specifications are consistent and extensive testing of the implementation programs will yield a highly bug free implementation for KSOS. However, nothing short of testing every possible case, a task that is clearly impossible, can guarantee completely bug-free programs. The only way to guarantee correct programs is by formal verification. Unfortunately, a complete formal verification of the KSOS programs (that is, the kernel and trusted NKSP software, plus guarantees of isolation of the rest of the system) within the time frame of the KSOS project would be very costly and must be considered beyond the state of the art -- particularly because of its dependence on including the hardware in the proof path. Further research is necessary in the verification certain types of programming constructs, e.g., parallelism. Also, further research is necessary in the area of automated theorem proving which, when better developed, will bring the cost of formal verification down to more reasonable levels. The technology necessary for a complete formal verification of KSOS programs is probably only a few years away, but is not available for this Phase II.

Even though complete formal verification is not practical, significant benefits can be derived from a formal verification of some of the KSOS programs. Formal verification is likely to discover bugs overlooked by informal checks and by extensive testing. Experience has shown that formal verification techniques find bugs in even small programs that were overlooked by experienced programmers. In fact, if the formal verification is begun early enough, it could pay for itself by disclosing bugs that would have to be corrected in a more costly fashion at some later stage. Also, formal verification of some of the KSOS system programs will demonstrate the feasibility of and the utility of performing a complete formal verification of all the programs at some later time.

PLAN FOR FORMAL PROGRAM VERIFICATION

In order to obtain the maximum benefits from the formal verification of a meaningful subset of the KSOS system, the programs on which formal verification is to be attempted must be carefully chosen. The programs to be verified must meet two criteria:

1. Formal verification of the program must be within the state of the art and not be inordinately costly.

2. The programs must be critical to the security of the system.

In order to determine which programs best meet this criteria, we will construct two lists of all the programs: the first will be in order of decreasing ease of formal verification and the second will be in order of decreasing importance to system security. For example, on the first list the program that moves files to and from the disk would appear after the program that returns amount of time used by a process because the former would contain more parallel constructs than the latter and would be more difficult to verify. On the second list the process scheduling program would appear after the process dispatching program because a bug in the dispatching program could result in a much greater security breach than a bug in the scheduler. The programs to be verified will be those closest to the top of both lists. We will chose as many programs as we feel can be formally verified within the allotted resources.

FURTHER IMPLEMENTATION PROOF TOOL CONSIDERATIONS

Automated formal verification systems consists of two major parts: the verification condition generator and the theorem prover. These two parts are analogous to the theorem generator and the theorem prover of the design proof tools. The verification condition generator (VCG) takes as in input the program to be proved and the formal specification of that program. The VCG generates the theorems whose correctness implies the correctness of the program. In order to do this, the VCG must have knowledge of the formal syntax and semantics of the language in which the program is written and knowledge of the syntax and semantics of the language in which the specification is written. The theorem prover, of course, attempts to prove the theorems generated by the VCG.

As noted above, several formal verification systems exist outside of SRI including those developed by Good et al. at the University of Texas, London et al. at USC Information Sciences Institute, and Luckham et al. at Stanford. At SRI there are a formal verification system developed by Elspas et al. and a theorem prover developed by Boyer and Moore. There is as yet no strong evidence to indicate that any one of these verification systems is far superior to any of the others. It is possible that any one could be adapted for KSOS. The adaptation requires incorporating knowledge of the KSOS Programming Language and SPECIAL into the VCG. Integrating one of these verification systems with the Boyer-Moore theorem prover should create a more powerful system. Since SRI has all the in-house expertise necessary to integrate and adapt its own systems, and since much of this work may very well be done anyway under other projects, the best approach seems to be to adapt the SRI tools to aid in the formal program verification. Since most of the desired tools are expected to be available by the time the verification is to begin, and the only significant tool development task will be to adapt the VCG to contain knowledge of the KSOS Programming language (a straightforward task), most of the resources for verification will go into the actual verification and only a modest amount into the tool development. The tools are essential, however, to making the formal verification tractable and creditable.

SECTION VI
TESTING

FACC views testing as the natural complement to the formal verification efforts. Testing can also expose classes of errors that are not addressed by formal verification, such as errors in the specifications. Testing is essential to the eventual success of KSOS. Since formal verification and testing are so closely related, it is appropriate to include FACC's plans for KSOS testing here.

FACC intends to provide three phases of testing for KSOS,

* module tests,
* partial integration tests (referred to as thread tests), and
* system tests.

Module testing is intended to assure that individual modules meet their specifications, both the formal specifications in SPECIAL and the programming specifications (Types B5 and C5). The most important product of the Phase I effort is a detailed and comprehensive set of specifications for KSOS. The Hierarchical Design Methodology (HDM) requires that each level in the design be accurately and completely specified in a precise, non-procedural language (for KSOS, SPECIAL). These formal specifications are complemented by the B5 Development Specifications and the C5 Product Specifications. All these specifications taken together represent a very thorough description of exactly what the module is supposed to do. Module testing, then, demonstrates that the as-built code does indeed realize its specifications.

Partial integration testing (which FACC calls "thread testing") is an attempt to provide early visibility to major system functions. In thread testing an attempt is made to test major system functions well before the entire system is integrated, or in many cases even implemented. Thread testing is a variant of top-down integration using stubs. Rather than stubbing out everything below some level, thread testing uses a complete software path from the highest level to the lowest. Those sections of modules that are not needed in a particular thread test may be stubbed. Thread testing exercises the higher-level modules more completely than can be done with pure top-down integration because the "stubs" are portions of the actual deliverable code. Thread testing also exposes certain types of interference between modules earlier than pure top-down integration. Within FACC thread testing has been employed with great success on a large (220,000 lines) software project.

System testing is primarily the pre-acceptance and acceptance testing for the KSOS product. The main components of system testing are:

* internal pre-acceptance tests,

* formal acceptance tests, including Category I and II tests,
* Functional Configuration Audit, and
* Physical Configuration Audit.

The nature of these tests is largely dictated by the contract and long-standing, Government-approved FACC quality assurance procedures and standards. FACC intends to use the system testing also as a demonstration of certain security features, such as resource quotas, that are lacking in UNIX*tm.

In the remainder of this section each of these test phases will be discussed in more detail, particularly with respect to how FACC intends to proceed with each phase.

MODULE TESTING

FACC intends to thoroughly exercise each module before committing it to integration. While complete path testing is unrealistic (for many of the same combinatoric reason as the infeasibility of full code proofs), FACC does intend to exercise every statement at least once. To do this, FACC intends to instrument the programs, either manually or automatically to detect the execution of each of the basic blocks of the program. By reducing this data for all test case executions it will be possible to assure that each statement has be executed. Both choices for the programming language (Euclid and Modula) facilitate this by being completely block structured. The basic blocks of a program are thus easily found. Also facilitating the module testing are the steps taken to improve the verifiability of the code. Making a module easier to prove also makes it easier to test.

Since production compilers for the prime candidate languages are not yet available, it is not clear how much support will be available for testing. In particular, an efficient profiler such as that in the C compiler would be of great benefit for module testing. If such support is not available, FACC intends to develop the minimally required support tools under the contract. On going internally funded (IRAD and capital) efforts will also supplement this tool development.

Administrative mechanisms will be used to assure that the testing has been accomplished. Copies (on-line) of the testing input and output will be reviewed before accepting the module for integration. This review is part of the larger first level internal quality assurance inspection. The other parts are as follows.

* A detailed review of the source code for style and aesthetics (e.g. naming conventions, comments, readability, etc.).

* A check of the module against its formal and programming specifications. This check will verify that every exception

cond...ion is present as an explicit test, and that the module implements its specifications. In these reviews, particular emphasis will be placed on boundary conditions for parameters and shared variables. Typically, it is the parameter which is just out of range that causes the more subtle errors.

These measures are mandatory for all deliverable software. They are separate and distinct from any formal proof efforts.

PARTIAL INTEGRATION (THREAD) TESTING

Early in Phase II FACC will review the KSOS design and define the threads and their associated tests. (Based on experience with much larger projects, this is a very modest task.) The thread definitions will also provide input to the more detailed scheduling and manpower allocation for Phase II.

Each thread consists of a sequence of inputs and anticipated outputs. One can expect that some of the early threads may require "switch flipping" to verify their performance. Other threads may require modest amounts of test fixtures such as dummy files etc. Normally, thread descriptions are developed with close cooperation between the implementation and test personnel. On a small project like KSOS, there will probably not be a separately identified test team until late in the integration effort. Rather, selected implementation personnel will also serve as the internal test team. By using formally identified threads, the potential "role conflict" here is minimized. Also FACC intends to try to keep the test and implementation personnel for a given function disjoint. Naturally, as more of the system is integrated this separation may lessen.

Perhaps the most difficult part of thread testing is error conditions and failures. Many of these conditions are initially triggered by hardware detected error conditions, such as device errors or "impossible" software conditions. To adequately test KSOS, these conditions must be simulated and the error handling mechanisms exercised. At present, FACC intends to exercise device error handling by slightly altering the real device drivers so that their device registers are directed elsewhere. By adding a test fixture to the security kernel that fills in these pseudo-device registers with an arbitrary, program controlled bit pattern, all the error handling aspects of device control can be exercises. It should be noted that this mechanism assumes that reasonable test cases can be generated that accurately reflect the way in which the hardware (mis-)behaves. This technique will allow all the system's devices to be bootstrapped up.

Internal software inconsistencies are more difficult because there are so many more possibilities. The HDM does provide some help by identifying both exceptions and assumptions. In partial integration testing many of the assumptions (SPECIAL ASSERT clauses) will actually be tested for. Standard error reporting mechanisms will be used when an assumption is violated. If the performance penalty of this additional checking is not too great, it might be well advised to leave it in the deliverable code.

KSOS will include certain types of checking not now present in UNIX*tm, particularly in the area of resource quotas. Partial integration testing will examine system behavior as these quotas are approached.

SYSTEM TESTS

System tests are intended to exercise the system as a whole and to demonstrate its behavior in as realistic a setting as possible. Virtually no test/jig software is included in the system although specially constructed files may be used. The system tests include internal pre-acceptance tests, formal acceptance tests, and the audits of MIL-STD-1521A on the as-built software.

The pre-acceptance tests blend with the latter stages of partial integration testing. For example, a thread test for login/logout exercises a large amount of the KSOS system. Also included in the pre-acceptance testing is a limited amount of benchmark timings. By running nearly identical tests on a UNIX*tm system , a basis for comparison can be established. The pre-acceptance testing will be performed by project personnel. Written objectives, procedures and results will be collected.

The formal acceptance testing form and content is largely dictated by the contract and existing, Government-approved quality assurance procedures. Project personnel will provide technical input to the support organizations who will actually perform and witness the tests FACC will make maximum use of automated tools to run and analyze the tests. These tools include shell files running on KSOS and if possible another machine simulating the terminal load on KSOS. Supervision and witnessing of the tests will be performed by the FACC quality assurance organization. Due to the unique requirements of KSOS versus other software products, close cooperation between quality assurance and project personnel is anticipated.

The final phase of acceptance testing are the audits of the as-built software. The guidelines of MIL-STD-1521A are sufficiently flexible to accommodate the requirements of KSOS. In particular, the Functional Configuration Audit can be used for a formal review of the design versus the mathematical model. The Physical Configuration Audit will be the vehicle for a review of the as-built code versus its specifications, both formal, and the C5 Product Specifications. The audits will also be the appropriate place for formal review of the mathematical proofs of correctness discussed elsewhere in this document.

SUMMARY

FACC feels that formal verification and testing are different facets of the same goal, providing the Government with a product of very high quality. To meet its requirements, KSOS may become perhaps the most thoroughly analyzed major software product ever produced. The testing program described in this section and the formal verification efforts described in other sections will in FACC's opinion satisfy the design requirements for KSOS.

DAT

ILM